```
// Computer Program Listing Appendix Under 37 CFR 1.52(e)
// client_info.h
// Copyright (c) 2004, Alio TV. All Rights Reserved.
#ifndef CLIENT_INFO_H
#define CLIENT_INFO_H
#include <alio_session.h>
#include <alio_error.h>
#include <data_object.h>
#include <array.h>
#include "time.h"
#include <client_media_list.h>
class AlioSession;
class ClientInfo: public DataObject
{
  public:
    static AlioError TableConfirm( AlioSession* alioSession );
    static AlioError TableDrop( AlioSession* alioSession );
   static AlioError SamplesCreate( AlioSession* alioSession, int base, int count );
    static ClientInfo* Find( AlioSession* alioSession, int id );
    static Array< ClientInfo* >* FindByContactTimeout( Array< ClientInfo *>* results, AlioSession* alioSession, long
long time );
    static Array< ClientInfo* >* FindTopPriorityDownload( Array< ClientInfo *>* results, AlioSession* alioSession );
    static Array< ClientInfo* >* FindNormalPriorityDownload( Array< ClientInfo *>* results, AlioSession* alioSession );
    static Array< ClientInfo* >* FindMediaSource( Array< ClientInfo *>* results, AlioSession* alioSession, int
mediaItemId, bool p2pEnable, bool serverEnable, bool avoidPassive );
    static Array< ClientInfo* >* FindServers( Array< ClientInfo *>* results, AlioSession* alioSession );
    ClientInfo( AlioSession* alioSessionInit );
    ClientInfo( AlioSession* alioSessionInit, int idInit );
    ~ClientInfo();
    AlioError createDBObject();
    AlioError deleteDBObject();
    AlioError updateDBObject();
    AlioError loadDBObject( char** row, int fieldCount );
    void contactTimeoutProcess( long long time );
    void mediaProcess( long long time );
    void ipAddressSet( const char* ipAddressInit );
    char* ipAddressGet();
    void ipPortSet( int ipPortInit );
    int ipPortGet();
    void ipMessagePortSet( int ipMessagePortInit ) { ipMessagePort = ipMessagePortInit; dirtySet(); }
    int ipMessagePortGet() { return ipMessagePort; }
    void ipTransferPortSet( int ipTransferPortInit ) { ipTransferPort = ipTransferPortInit; dirtySet(); }
    int ipTransferPortGet() { return ipTransferPort; }
    void customerIdSet( int customerIdInit ) { customerId = customerIdInit; dirtySet(); }
    int customerIdGet() { return customerId; }
    void uplinkCapacitySet( int uplinkCapacityInit );
    int uplinkCapacityGet();
    void uplinkCurrentSet( int uplinkCurrentInit );
    int uplinkCurrentGet();
    void uplinkLastSet( long long uplinkLastInit ) { uplinkLast = uplinkLastInit; dirtySet(); }
```

```
    long long uplinkLastGet() { return uplinkLast; }
    void downlinkCapacitySet( int downlinkCapacityInit );
    int downlinkCapacityGet();
    void downlinkCurrentSet( int downlinkCurrentInit );
    int downlinkCurrentGet();
    void downlinkLastSet( long long downlinkLastInit ) { downlinkLast = downlinkLastInit; dirtySet(); }
    long long downlinkLastGet() { return downlinkLast; }
    void storageCapacitySet( long long storageCapacityInit );
    long long storageCapacityGet();
    void storageCurrentSet( long long storageCurrentInit );
    long long storageCurrentGet();
    void mediaUnwatchedCountSet( int mediaUnwatchedCountInit ) { mediaUnwatchedCount =
mediaUnwatchedCountInit; dirtySet(); }
    int mediaUnwatchedCountGet() { return mediaUnwatchedCount; }
    void mediaAvailableCountSet( int mediaAvailableCountInit ) { mediaAvailableCount = mediaAvailableCountInit;
dirtySet(); }
    int mediaAvailableCountGet() { return mediaAvailableCount; }
    void mediaListSizeSet( int mediaListSizeInit ) { mediaListSize = mediaListSizeInit; dirtySet(); }
    int mediaListSizeGet() { return mediaListSize; }
    void mediaOutTimeSet( long long mediaOutTimeInit ) { mediaOutTime = mediaOutTimeInit; dirtySet(); }
    long long mediaOutTimeGet() { return mediaOutTime; }
    void mediaRequestedTimeSet( long long mediaRequestedTimeInit ) { mediaRequestedTime =
mediaRequestedTimeInit; dirtySet(); }
    long long mediaRequestedTimeGet() { return mediaRequestedTime; }
    void mediaTouchedSet( bool mediaTouchedInit ) { mediaTouched = mediaTouchedInit; dirtySet(); }
    bool mediaTouchedGet() { return mediaTouched; }
    void connectedSet( bool connectedInit );
    bool connectedGet();
    void contactLastSet( long long contactLastInit );
    long long contactLastGet();
    void contactTimeoutSet( long long contactTimeoutInit );
    long long contactTimeoutGet();
    void passiveSet( bool passiveInit );
    bool passiveGet();
    void preloadingSet( bool preloadingInit ) { preloading = preloadingInit; dirtySet(); }
    bool preloadingGet() { return preloading; }
    void serverSet( bool serverInit ) { server = serverInit; dirtySet(); }
    bool serverGet() { return server; }
    void simulatedSet( bool simulatedInit ) { simulated = simulatedInit; dirtySet(); }
    bool simulatedGet() { return simulated; }
    int mediaCountGet(){ mediaListConfirm(); return mediaList->sizeGet(); }
    int mediaItemForDownloadNextGet();
    ClientMedia* clientMediaNew( int mediaId, bool requested, int rank );
    ClientMedia* clientMediaGetFromMediaId( int mediaId );
    void clientMediaComplete( int mediaId );
    int clientMediaIdForMediaIdGet( int mediaInfoId );
    void clientMediaMove( int rankOld, int rankNew );
    void clientMediaRemove( int rank );
    void clientMediaAdd( int rank, int mediaId );
  private:
```

```
    AlioSession* alioSession;
    void processMessage( Message* message );
    void processMessageClientNewConfirm( Array< char* >* arguments );
    void init();
    static Array< ClientInfo* >* FindByQuery( Array< ClientInfo *>* results, AlioSession* alioSession, char* query );
    static ClientInfo* FindFromCache( AlioSession* alioSession, int id );
    void mediaListConfirm();
    long long timeGet() { return time( 0 ); }
    // Kind of database values
    ClientMediaList* mediaList;
    // DATABASE VALUES
    bool  server;
    char* ipAddress;
    int   ipTransferPort;
    int   ipMessagePort;
    int   customerId;
    int   uplinkCapacity;
    int   uplinkCurrent;
    long long  uplinkLast;
    int   downlinkCapacity;
    int   downlinkCurrent;
    long long  downlinkLast;
    long long  storageCapacity;
    long long  storageCurrent;
    int   mediaUnwatchedCount;
    int   mediaAvailableCount;
    int   mediaListSize;
    long long  mediaOutTime;
    long long  mediaRequestedTime;
    bool  mediaTouched;
    bool  connected;
    long long  contactLast;
    long long  contactTimeout;
    bool  preloading;
    bool  passive;
    bool simulated;
};
#endif
// scheduler.h
// Copyright (c) 2004, Alio TV. All Rights Reserved.
#ifndef SCHEDULER_H
#define SCHEDULER_H
#include "scheduleruiinterface.h"
#include "schedulerinterface.h"
#include <unistd.h>
#include <pthread.h>
#include <alio_session.h>
#include <log.h>
#include <media_info.h>
#include <message.h>
```

```
#include <client_info.h>
#include <client_media.h>
#include <message_engine.h>
#include <transfer.h>
#include <customer.h>
#include <customer_media.h>
/**
   Main Class for the Scheduler
  **/
class Scheduler: public SchedulerInterface
{
  public:
    Scheduler( int init );
    virtual ~Scheduler() {}
    void uiInterfaceSet( SchedulerUIInterface* uiInterfaceInit );
    // SchedulerInterface
    void run();
    void stop();
    long long timeGet();
    void ipPortSet( int ipPortInit ) { ipPort = ipPortInit; }
    void p2pEnableSet( bool enable );
    void serverEnableSet( bool enable );
    void threadRunning( );
  private:
    void iterate( long long secs );
    void processMessage( Message* message );
    void processMessageClientNew( Array< char* >* arguments );
    void processMessageContact( int fromId, Array< char* >* arguments );
    void processMessagePing( int fromId, Array< char* >* arguments );
    void processMessageClientMediaNew( int fromId, Array< char* >* arguments );
    void processMessageClientMediaComplete( int fromId, Array< char* >* arguments );
    void processMessageTransferStarting( int fromId, Array< char* >* arguments );
    void processMessageTransferListening( int fromId, Array< char* >* arguments );
    void processMessageTransferConnecting( int fromId, Array< char* >* arguments );
    void processMessageTransferTransferring( int fromId, Array< char* >* arguments );
    void processMessageTransferProgress( int fromId, Array< char* >* arguments );
    void processMessageTransferComplete( int fromId, Array< char* >* arguments );
    void processMessageTransferTimeout( int fromId, Array< char* >* arguments );
    void processMessageTransferError( int fromId, Array< char* >* arguments );
    void processMessageMediaAuthorizationRequest( int fromId, Array< char* >* arguments );
    void processMessageClientMediaMove( int fromId, Array< char* >* arguments );
    void processMessageClientMediaAdd( int fromId, Array< char* >* arguments );
    void processMessageClientMediaRemove( int fromId, Array< char* >* arguments );
    void messagesMonitor( long long time );
    void mediaItemRemove( int clientId, int rank );
    void mediaItemMove( int clientId, int rankOld, int rankNew );
    void mediaItemAdd( int clientId, int rank, int mediaId );
    bool transfersDesign( bool topPriority );
    void transfersMonitor( long long time );
    void transferAbort( Transfer* transfer );
```

```
    Transfer* transferStateSet( int fromId, Transfer::State state, Array< char* >* arguments );
    void transferAbortMessageSend( int clientId, int transferId );
    void databaseInitialize( int init );
    void customerNewCreate( );
    void timeSubtract( struct timeval* diff, struct timeval* start, struct timeval* end );
    long long randomTime();
    AlioSession alioSession;
    SchedulerUIInterface* uiInterface;
    MessageEngine* messageEngine;
    pthread_t runningThread;
    pthread_mutex_t runningMutex;
    bool running;
    long long customerSpawnLast;
    int customerCount;
    int ipPort;
    bool p2pEnable;
    bool serverEnable;
    bool transferPriority;
};
#endif
// transfer_engine.h
// Copyright (c) 2004, Alio TV. All Rights Reserved.
#ifndef TRANSFER_ENGINE_H
#define TRANSFER_ENGINE_H
#include <alio.h>
#include <alio_session.h>
#include <transfer_engine_interface.h>
#include <transfer.h>
#include <log.h>
#include <array.h>
#include <unistd.h>
#include <pthread.h>
/**
    Main Class for the TransferEngine
   **/
#define TRANSFER_ENGINE_QUEUED_CONNECTIONS      5  // what's this for?
#define MAX_BUF_SIZE                        4096  // limit might be intmax
#define CONNECT_RETRIES_MAX                   10
#define BIND_RETRIES_MAX                       5
#define CONNECT_RETRY_SLEEP_SECONDS            5  // 0 means don't sleep
#define BIND_RETRY_SLEEP_SECONDS               1  // 0 means don't sleep
#define TRANSFER_UNIT_SIZE                  4096  // limit might be intmax
#define FILE_CREATE_MODE                    0666  // rw for all
#define ACCEPT_TIMEOUT_SECONDS                30
#define ACCEPT_TIMEOUT_MILLISECONDS            0
class TransferEngine;
class TransferRecord
{
  public:
    TransferRecord( TransferEngine* transferEngine,
```

```
                int schedulerTransferId,
                int localId,
                int localTransferPort,
                int localClientMediaId,
                int remoteId,
                char* remoteIPAddress,
                int remoteTransferPort,
                bool localActive, bool localSends,
                char* localPath,
                long long start,
                long long length,
                bool virtualTranfer );
    ~TransferRecord();
    TransferEngine* transferEngine;
    int schedulerTransferId;
    int localId;
    int localTransferPort;
    int localClientMediaId;
    int remoteId;
    char* remoteIPAddress;
    int remoteTransferPort;
    bool localActive;
    bool localSends;
    char* localPath;
    long long start;
    long long length;
    long long current;
    bool virtualTransfer;
    pthread_t thread;
};
class TransferEngine: public TransferEngineInterface
{
  public:
    TransferEngine( int idInit, const char* databaseName );
    virtual ~TransferEngine() { }
    // Transfer Engine Interface
    virtual void callbackInterfaceSet( TransferEngineCallbackInterface* transferEngineCallbackInterfaceInit )
      { transferEngineCallbackInterface = transferEngineCallbackInterfaceInit; }
    virtual void pathBaseSet( const char* pathBaseInit );
    virtual void transferStart( int schedulerId,
                                int localId,
                                int localTransferPort,
                                int localClientMediaId,
                                int remoteId,
                                char* remoteIPAddress,
                                int remoteTransferPort,
                                bool localActive, bool localSends,
                                char* localPath,
                                long long start,
                                long long length,
```

```
                        bool virtualTransfer );
    virtual void transferCancel( int schedulerId );
    void threadRunning( TransferRecord* transferRecord );
  private:
    void databaseConnectionInitialize( const char* database );
    long long timeGet();
    int acceptWithTimeout( int                serverSocket,
                        struct sockaddr_in* clientInternetAddress,
                        struct timeval      timeout);
    Array< TransferRecord *> transferRecords;
    pthread_mutex_t controlMutex;
    int id;
    TransferEngineCallbackInterface* transferEngineCallbackInterface;
    AlioSession alioSession;
    pthread_t senderThread;
    char* pathBase;
};
#endif
// Computer Program Listing Appendix Under 37 CFR 1.52(e)
// client_info.cpp
// Copyright (c) 2004, Alio TV. All Rights Reserved.
#include <client_info.h>
#include <time.h>
#include <log.h>
#define CLIENT_INFO_QUERY_MAXSIZE       1000
#define CLIENT_INFO_TABLE_NAME          "client_info"
#define CLIENT_INFO_TABLE_FIELD_COUNT    26
#define CLIENT_CONTACT_TIMEOUT_BASE      300
#define CLIENT_CONTACT_TIMEOUT_VARIANCE  60
#define CLIENT_INFO_FREESPACE           2000000000LL
#define CLIENT_INFO_WATCHABLE_MAX        20
#define CLIENT_INFO_NORMS_MAX            100
#define CLIENT_INFO_HIGH_MAX            1000
AlioError ClientInfo::TableConfirm( AlioSession* alioSession )
{
  AlioError error;
  error = alioSession->execute( "SELECT * FROM "
                        CLIENT_INFO_TABLE_NAME
                        " LIMIT 1;" );
  bool create = ( error != ALIO_OK );
  if ( error == ALIO_OK )
  {
    int fields;
    alioSession->resultsGet();
    alioSession->resultFieldCountGet( &fields );
    if ( fields != CLIENT_INFO_TABLE_FIELD_COUNT )
    {
      alioSession->dropTable( CLIENT_INFO_TABLE_NAME );
      create = true;
    }
```

```
    alioSession->resultsFree();
  }
  if ( create )
  {
    error = alioSession->execute( "CREATE TABLE " CLIENT_INFO_TABLE_NAME
                                  "  ( id INT PRIMARY KEY,"
                                  "    ip_address VARCHAR( 100 ),"
                                  "    ip_message_port INT,"
                                  "    ip_transfer_port INT,"
                                  "    customer_id INT,"
                                  "    uplink_capacity INT,"
                                  "    uplink_current INT,"
                                  "    uplink_last INT,"
                                  "    downlink_capacity INT,"
                                  "    downlink_current INT,"
                                  "    downlink_last INT,"
                                  "    storage_capacity BIGINT,"
                                  "    storage_current BIGINT,"
                                  "    media_unwatched_count INT,"
                                  "    media_available_count INT,"
                                  "    media_list_size INT,"
                                  "    media_out_time BIGINT,"
                                  "    media_requested_time BIGINT,"
                                  "    media_touched INT,"
                                  "    connected INT,"
                                  "    contact_last BIGINT,"
                                  "    contact_timeout BIGINT,"
                                  "    passive INT, "
                                  "    preloading INT, "
                                  "    server INT, "
                                  "    simulated INT,"
                                  "    INDEX i1 ( media_unwatched_count ),"
                                  "    INDEX i2 ( downlink_last ),"
                                  "  );" );
    if ( error != ALIO_OK )
      return error;
  }
  return ALIO_OK;
}
AlioError ClientInfo::TableDrop( AlioSession* alioSession )
{
  alioSession->dropTable( CLIENT_INFO_TABLE_NAME );
  alioSession->idReset( CLIENT_INFO_TABLE_NAME );
  return ALIO_OK;
}
AlioError ClientInfo::SamplesCreate( AlioSession* alioSession, int base, int count )
{
  alioSession->transactionStart();
  for ( int i = 0; i < count; i++ )
  {
```

```
    char ip[ 100 ];
    ClientInfo *ci = new ClientInfo( alioSession );
    sprintf( ip, "www.ipaddress_%d_%d.com", base, i );
    ci->ipAddressSet( ip );
    ci->ipMessagePortSet( 10000 );
    ci->ipMessagePortSet( 10001 );
    ci->ipTransferPortSet( 10001 );
    ci->customerIdSet( i );
    ci->uplinkCapacitySet( 128000 );
    ci->uplinkCurrentSet( 0 );
    ci->uplinkLastSet( 0 );
    ci->downlinkCapacitySet( 1000000 );
    ci->downlinkCurrentSet( 0 );
    ci->downlinkLastSet( 0 );
    ci->storageCapacitySet( 40000000 );
    ci->storageCurrentSet( 0 );
    ci->connectedSet( false );
    ci->contactLastSet( 0 );
    ci->mediaUnwatchedCountSet( 0 );
    ci->mediaAvailableCountSet( 0 );
    ci->mediaListSizeSet( 0 );
    ci->mediaTouchedSet( true );
    ci->mediaOutTimeSet( 0 );
    ci->mediaRequestedTimeSet( 0 );
    ci->contactTimeoutSet( time( 0 ) + ( rand() % 60 ) + ( rand() % 60 )  );
    ci->passiveSet( false );
  }
  alioSession->transactionEnd();
  return ALIO_OK;
}
ClientInfo* ClientInfo::Find( AlioSession* alioSession, int id )
{
  // first need to see if the object is already loaded
  ClientInfo* ci;
  if ( ( ci = FindFromCache( alioSession, id ) ) )
    return ci;
  char q[ CLIENT_INFO_QUERY_MAXSIZE ];
  snprintf( q, CLIENT_INFO_QUERY_MAXSIZE, "SELECT * from " CLIENT_INFO_TABLE_NAME " WHERE id =
%d;", id );
  AlioError status;
  status = alioSession->execute( q );
  if ( status != ALIO_OK )
    return 0;
  status = alioSession->resultsGet();
  if ( status != ALIO_OK )
    return 0;
  long long count;
  alioSession->resultCountGet( &count );
  if ( count == 1 )
  {
```

```
    ci = new ClientInfo( alioSession, id );
    int fieldCount;
    alioSession->resultFieldCountGet( &fieldCount );
    AlioError status = ci->loadDBObject( alioSession->resultRowGet(), fieldCount );
    alioSession->resultsFree();
    if ( status == ALIO_OK )
    {
      //if ( alioSession->transactionActive() )
      //  alioSession->clientInfosGet()->append( mi );
      return ci;
    }
    else
     return 0;
  }
  else
  {
    alioSession->resultsFree();
    return 0;
  }
}
Array< ClientInfo * >* ClientInfo::FindByContactTimeout( Array< ClientInfo *>* results, AlioSession* alioSession, long
long time  )
{
  char q[ CLIENT_INFO_QUERY_MAXSIZE ];
  snprintf( q, CLIENT_INFO_QUERY_MAXSIZE, "SELECT * from " CLIENT_INFO_TABLE_NAME " WHERE
contact_timeout < %lld;", time );
  return FindByQuery( results, alioSession, q );
}
Array< ClientInfo* >* ClientInfo::FindTopPriorityDownload( Array< ClientInfo *>* results, AlioSession* alioSession )
{
  char q[ CLIENT_INFO_QUERY_MAXSIZE ];
  snprintf( q, CLIENT_INFO_QUERY_MAXSIZE, "SELECT * from " CLIENT_INFO_TABLE_NAME
                                  "  WHERE media_unwatched_count = 0 AND"
                                  "        media_list_size > 0 AND "
                                  "        downlink_current = 0 AND "
                                  "        connected = 1 AND "
                                  "        server = 0 AND "
                                  "        storage_capacity - storage_current > %lld "
                                  "  ORDER BY media_out_time"
                                  "  LIMIT %d ;", CLIENT_INFO_FREESPACE, CLIENT_INFO_HIGH_MAX );
  return FindByQuery( results, alioSession, q );
}
Array< ClientInfo* >* ClientInfo::FindNormalPriorityDownload( Array< ClientInfo *>* results, AlioSession* alioSession )
{
  char q[ CLIENT_INFO_QUERY_MAXSIZE ];
  snprintf( q, CLIENT_INFO_QUERY_MAXSIZE, "SELECT * from " CLIENT_INFO_TABLE_NAME
                                  "  WHERE downlink_current = 0 AND "
                                  "        server = 0 AND "
                                  "        media_available_count < %d AND "
                                  "        media_list_size > media_available_count AND "
```

```
                              "       connected = 1 AND "
                              "       storage_capacity - storage_current > %lld "
                              "  ORDER BY downlink_last"
                              "  LIMIT %d ;", CLIENT_INFO_WATCHABLE_MAX, CLIENT_INFO_FREESPACE,
CLIENT_INFO_NORMS_MAX );
  return FindByQuery( results, alioSession, q );
}
Array< ClientInfo* >* ClientInfo::FindMediaSource( Array< ClientInfo *>* results, AlioSession* alioSession, int
mediaItemId, bool p2pEnable, bool serverEnable, bool avoidPassive )
{
  char q[ CLIENT_INFO_QUERY_MAXSIZE ];
  char *tweak1;
  char *tweak2;
  if ( p2pEnable )
  {
    if ( serverEnable )
      tweak1 = "";
    else
      tweak1 = " AND " CLIENT_INFO_TABLE_NAME ".server = 0 ";
  }
  else
  {
    if ( serverEnable )
      tweak1 = " AND " CLIENT_INFO_TABLE_NAME ".server = 1 ";
    else
    {
      results->empty();
      return results;
    }
  }
  if ( avoidPassive )
    tweak2 = " AND " CLIENT_INFO_TABLE_NAME ".passive = 0 ";
  else
    tweak2 = "";
  snprintf( q, CLIENT_INFO_QUERY_MAXSIZE, "SELECT " CLIENT_INFO_TABLE_NAME ".* from "
CLIENT_INFO_TABLE_NAME " , " CLIENT_MEDIA_TABLE_NAME
                              "  WHERE " CLIENT_INFO_TABLE_NAME ".id = " CLIENT_MEDIA_TABLE_NAME
".client_id AND"
                              "       " CLIENT_MEDIA_TABLE_NAME ".media_id = %d AND"
                              "       " CLIENT_MEDIA_TABLE_NAME ".complete = 1 AND"
                              "       " CLIENT_INFO_TABLE_NAME ".uplink_current < uplink_capacity"
                              "       %s "
                              "       %s "
                              "  ORDER BY " CLIENT_INFO_TABLE_NAME ".uplink_last "
                              "  LIMIT 1;",
                              mediaItemId,
                              tweak1,
                              tweak2 );
  return FindByQuery( results, alioSession, q );
}
```

```
Array< ClientInfo* >* ClientInfo::FindServers( Array< ClientInfo *>* results, AlioSession* alioSession )
{
  char q[ CLIENT_INFO_QUERY_MAXSIZE ];
  snprintf( q, CLIENT_INFO_QUERY_MAXSIZE, "SELECT * from " CLIENT_INFO_TABLE_NAME
                                    "  WHERE server = 1"
                                    "    AND connected = 1;" );
  return FindByQuery( results, alioSession, q );
}
Array< ClientInfo * >* ClientInfo::FindByQuery( Array< ClientInfo *>* results, AlioSession* alioSession, char* q  )
{
  AlioError status;
  status = alioSession->execute( q );
  if ( status != ALIO_OK )
    return results;
  status = alioSession->resultsGet();
  if ( status != ALIO_OK )
    return results;
  long long count;
  alioSession->resultCountGet( &count );
  int fieldCount;
  alioSession->resultFieldCountGet( &fieldCount );
  if ( fieldCount == CLIENT_INFO_TABLE_FIELD_COUNT && count > 0 )
  {
    if ( results == 0 )
      results = new Array< ClientInfo *>;
    else
      results->empty();
    for ( int i = 0; i < count; i++ )
    {
      char** resultRow = alioSession->resultRowGet();
      int resultId;
      sscanf( resultRow[ 0 ], "%d", &resultId );
      ClientInfo* m = FindFromCache( alioSession, resultId );
      if ( m == 0 )
      {
        m = new ClientInfo( alioSession, resultId );
        m->loadDBObject( resultRow, fieldCount );
      }
      results->append( m );
    }
  }
  alioSession->resultsFree();
  return results;
}
ClientInfo* ClientInfo::FindFromCache( AlioSession* alioSession, int id )
{
  Array< ClientInfo* >* cs = alioSession->clientInfoGet();
  for ( int i = 0; i < cs->sizeGet(); i++ )
  {
    ClientInfo* c = cs->elementGet( i );
```

```
    if ( c->idGet() == id )
      return c;
  }
  return 0;
}
ClientInfo::ClientInfo( AlioSession* alioSessionInit ) : DataObject( alioSessionInit )
{
  int idNext;
  init();
  alioSession = alioSessionInit;
  alioSession->idGet( CLIENT_INFO_TABLE_NAME, 1, &idNext );
  idSet( idNext );
  if ( alioSession->transactionActive() )
    alioSession->clientInfoGet()->append( this );
}
ClientInfo::ClientInfo( AlioSession* alioSessionInit, int idInit ) : DataObject( alioSessionInit )
{
  init();
  alioSession = alioSessionInit;
  idSet( idInit );
  if ( alioSession->transactionActive() )
    alioSession->clientInfoGet()->append( this );
}
void ClientInfo::init()
{
  server = 0;
  ipAddress = 0;
  ipMessagePort = 0;
  ipTransferPort = 0;
  customerId = -1;
  uplinkCapacity = 0;
  uplinkCurrent = 0;
  uplinkLast = 0;
  downlinkCapacity = 0;
  downlinkCurrent = 0;
  downlinkLast = 0;
  storageCapacity = 0;
  storageCurrent = 0;
  connected = 0;
  contactLast = 0;
  contactTimeout = 0;
  passive = 0;
  mediaUnwatchedCount = 0;
  mediaAvailableCount = 0;
  mediaListSize = 0;
  mediaOutTime = 0;
  mediaRequestedTime = 0;
  mediaTouched = 0;
  mediaList = 0;
  preloading = true;
```

```
  simulated = false;
}
ClientInfo::~ClientInfo()
{
  if ( ipAddress != 0 )
    free( ipAddress );
  alioSession->deletingObject( this );
}
AlioError ClientInfo::createDBObject()
{
  char q[ CLIENT_INFO_QUERY_MAXSIZE ];
  snprintf( q, CLIENT_INFO_QUERY_MAXSIZE,
          "INSERT INTO " CLIENT_INFO_TABLE_NAME " SET"
          " id = %d,"
          " ip_address = \"%s\","
          " ip_message_port = %d, "
          " ip_transfer_port = %d, "
          " customer_id = %d, "
          " uplink_capacity = %d,"
          " uplink_current = %d,"
          " uplink_last = %lld,"
          " downlink_capacity = %d,"
          " downlink_current = %d,"
          " downlink_last = %lld,"
          " storage_capacity = %lld,"
          " storage_current = %lld,"
          " media_unwatched_count = %d,"
          " media_available_count = %d,"
          " media_list_size = %d,"
          " media_out_time = %lld,"
          " media_requested_time = %lld,"
          " media_touched = %d,"
          " connected = %d,"
          " contact_last = %lld,"
          " contact_timeout = %lld,"
          " passive = %d,"
          " preloading = %d,"
          " server = %d,"
          " simulated = %d ;",
          idGet(),
          ipAddress, ipMessagePort, ipTransferPort,
          customerId,
          uplinkCapacity, uplinkCurrent, uplinkLast,
          downlinkCapacity, downlinkCurrent, downlinkLast,
          storageCapacity, storageCurrent,
          mediaUnwatchedCount, mediaAvailableCount,
          mediaListSize,
          mediaOutTime,
          mediaRequestedTime,
          mediaTouched,
```

```
        connected, contactLast, contactTimeout,
        passive,
        preloading,
        server,
        simulated );
  // printf( "CLIENT INFO INSERT QUERY \n %s \n", q );
  return alioSession->execute( q );
}
AlioError ClientInfo::deleteDBObject()
{
  char q[ CLIENT_INFO_QUERY_MAXSIZE ];
  snprintf( q, CLIENT_INFO_QUERY_MAXSIZE,
        "DELETE FROM " CLIENT_INFO_TABLE_NAME " WHERE"
          " id = %d;", idGet() );
  return alioSession->execute( q );
}
AlioError ClientInfo::updateDBObject()
{
  char q[ CLIENT_INFO_QUERY_MAXSIZE ];
  snprintf( q, CLIENT_INFO_QUERY_MAXSIZE,
        "UPDATE " CLIENT_INFO_TABLE_NAME " SET"
        " ip_address = \"%s\","
        " ip_message_port = %d, "
        " ip_transfer_port = %d, "
        " customer_id = %d, "
        " uplink_capacity = %d,"
        " uplink_current = %d,"
        " uplink_last = %lld,"
        " downlink_capacity = %d,"
        " downlink_current = %d,"
        " downlink_last = %lld,"
        " storage_capacity = %lld,"
        " storage_current = %lld,"
        " media_unwatched_count = %d,"
        " media_available_count = %d,"
        " media_list_size = %d,"
        " media_out_time = %lld,"
        " media_requested_time = %lld,"
        " media_touched = %d,"
        " connected = %d,"
        " contact_last = %lld,"
        " contact_timeout = %lld,"
        " passive = %d, "
        " preloading = %d, "
        " server = %d,"
        " simulated = %d "
        " WHERE id = %d;",
        ipAddress,
        ipMessagePort,
        ipTransferPort,
```

```
        customerId,
        uplinkCapacity,
        uplinkCurrent,
        uplinkLast,
        downlinkCapacity,
        downlinkCurrent,
        downlinkLast,
        storageCapacity, storageCurrent,
        mediaUnwatchedCount, mediaAvailableCount,
        mediaListSize,
        mediaOutTime,
        mediaRequestedTime,
        mediaTouched,
        connected, contactLast, contactTimeout,
        passive,
        preloading,
        server,
        simulated,
        idGet() );
  return alioSession->execute( q );
}
AlioError ClientInfo::loadDBObject( char** row, int fieldCount )
{
  if ( fieldCount == CLIENT_INFO_TABLE_FIELD_COUNT )
  {
    long long idInit;
    sscanf( row[ 0 ], "%lld", &idInit );
    idSet( idInit );
    if ( ipAddress )
      free( ipAddress );
    if ( row[ 1 ] )
      ipAddress = strdup( row[ 1 ] );
    sscanf( row[ 2  ],  "%d", &ipMessagePort );
    sscanf( row[ 3  ],  "%d", &ipTransferPort );
    sscanf( row[ 4  ],  "%d", &customerId );
    sscanf( row[ 5  ],  "%d", &uplinkCapacity );
    sscanf( row[ 6  ],  "%d", &uplinkCurrent );
    sscanf( row[ 7  ],  "%lld", &uplinkLast );
    sscanf( row[ 8  ],  "%d", &downlinkCapacity );
    sscanf( row[ 9  ],  "%d", &downlinkCurrent );
    sscanf( row[ 10  ],  "%lld", &downlinkLast );
    sscanf( row[ 11  ],  "%lld", &storageCapacity );
    sscanf( row[ 12  ],  "%lld", &storageCurrent );
    sscanf( row[ 13  ],  "%d", &mediaUnwatchedCount );
    sscanf( row[ 14  ],  "%d", &mediaAvailableCount );
    sscanf( row[ 15  ],  "%d", &mediaListSize );
    sscanf( row[ 16  ], "%lld", &mediaOutTime );
    sscanf( row[ 17  ], "%lld", &mediaRequestedTime );
    int mediaTouchedInit;
    sscanf( row[ 18  ], "%d", &mediaTouchedInit );
```

```
    int connectedInit;
    sscanf( row[ 19  ], "%d", &connectedInit );
    connected = connectedInit != 0;
    sscanf( row[ 20 ], "%lld", &contactLast );
    sscanf( row[ 21 ], "%lld", &contactTimeout );
    int passiveInit;
    sscanf( row[ 22 ], "%d", &passiveInit );
    passive = passiveInit != 0;
    int preloadingInit;
    sscanf( row[ 23 ], "%d", &preloadingInit );
    preloading = preloadingInit != 0;
    int serverInit;
    sscanf( row[ 24 ], "%d", &serverInit );
    server = serverInit != 0;
    int simulatedInit;
    sscanf( row[ 25 ], "%d", &simulatedInit );
    simulated = simulatedInit != 0;
    cleanSet();
    return ALIO_OK;
  }
  else
    return ALIO_ERROR_BAD_VALUE;
}
void ClientInfo::mediaProcess( long long time )
{
  // load the client media
  mediaListConfirm();
  mediaUnwatchedCountSet( mediaList->mediaUnwatchedCountGet() );
  mediaAvailableCountSet( mediaList->mediaAvailableCountGet() );
  mediaListSizeSet( mediaList->sizeGet() );
  storageCurrentSet( mediaList->storageCurrentGet() );
  // assign a time at which the media count goes to zero
  if ( mediaUnwatchedCountGet() == 0 && mediaOutTimeGet() == 0 && mediaListSizeGet() > 0 )
    mediaOutTimeSet( time );
  mediaTouchedSet( false );
}
int ClientInfo::mediaItemForDownloadNextGet()
{
  mediaListConfirm();
  return mediaList->mediaItemForDownloadNextGet();
}
//
// INDIRECT GETTERS
//
void ClientInfo::mediaListConfirm()
{
  if ( mediaList == 0 )
    mediaList = new ClientMediaList( alioSession, idGet() );
}
ClientMedia* ClientInfo::clientMediaNew( int mediaId, bool requested, int rank )
```

```
{
  Log::L( alioSession, 3, "ClientCore", "New ClientMedia", "MediaId %d Requested %d Rank %d", mediaId, requested,
rank );
  mediaListConfirm();
  return mediaList->add( mediaId, requested, rank );
}
void ClientInfo::clientMediaComplete( int mediaId )
{
  Log::L( alioSession, 3, "ClientCore", "ClientMedia Complete", "MediaId %d", mediaId );
  mediaListConfirm();
  return mediaList->completeSet( mediaId );
}
int ClientInfo::clientMediaIdForMediaIdGet( int mediaInfoId )
{
  mediaListConfirm();
  return mediaList->clientMediaIdForMediaIdGet( mediaInfoId );
}
ClientMedia* ClientInfo::clientMediaGetFromMediaId( int mediaInfoId )
{
  mediaListConfirm();
  return mediaList->clientMediaGetFromMediaId( mediaInfoId );
}
void ClientInfo::clientMediaMove( int rankOld, int rankNew )
{
  mediaListConfirm();
  mediaList->move( rankOld, rankNew );
}
void ClientInfo::clientMediaRemove( int rank )
{
  mediaListConfirm();
  mediaList->remove( rank );
  mediaProcess( timeGet() );
}
void ClientInfo::clientMediaAdd( int rank, int mediaId )
{
  mediaListConfirm();
  mediaList->add(mediaId, true, rank );
  mediaProcess( timeGet() );
}
//
// GETTERS AND SETTERS
//
void ClientInfo::ipAddressSet( const char* ipAddressInit )
{
  if ( ipAddress != NULL )
    free( ipAddress );
  ipAddress = strdup( ipAddressInit );
  dirtySet();
}
char* ClientInfo::ipAddressGet()
```

```
{
  return ipAddress;
}
void ClientInfo::uplinkCapacitySet( int uplinkCapacityInit )
{
  uplinkCapacity = uplinkCapacityInit;
  dirtySet();
}
int ClientInfo::uplinkCapacityGet()
{
  return uplinkCapacity;
}
void ClientInfo::uplinkCurrentSet( int uplinkCurrentInit )
{
  uplinkCurrent = uplinkCurrentInit;
  dirtySet();
}
int ClientInfo::uplinkCurrentGet()
{
  return uplinkCurrent;
}
void ClientInfo::downlinkCapacitySet( int downlinkCapacityInit )
{
  downlinkCapacity = downlinkCapacityInit;
  dirtySet();
}
int ClientInfo::downlinkCapacityGet()
{
  return downlinkCapacity;
}
void ClientInfo::downlinkCurrentSet( int downlinkCurrentInit )
{
  downlinkCurrent = downlinkCurrentInit;
  dirtySet();
}
int ClientInfo::downlinkCurrentGet()
{
  return downlinkCurrent;
}
void ClientInfo::storageCapacitySet( long long storageCapacityInit )
{
  storageCapacity = storageCapacityInit;
  dirtySet();
}
long long ClientInfo::storageCapacityGet()
{
  return storageCapacity;
}
void ClientInfo::storageCurrentSet( long long storageCurrentInit )
{
```

```
  storageCurrent = storageCurrentInit;
  dirtySet();
}
long long ClientInfo::storageCurrentGet()
{
  return storageCurrent;
}
void ClientInfo::connectedSet( bool connectedInit )
{
  connected = connectedInit;
  dirtySet();
}
bool ClientInfo::connectedGet()
{
  return connected;
}
void ClientInfo::contactLastSet( long long contactLastInit )
{
  contactLast = contactLastInit;
  dirtySet();
}
long long ClientInfo::contactLastGet()
{
  return contactLast;
}
void ClientInfo::contactTimeoutSet( long long contactTimeoutInit )
{
  contactTimeout = contactTimeoutInit;
  dirtySet();
}
long long ClientInfo::contactTimeoutGet()
{
  return contactTimeout;
}
void ClientInfo::passiveSet( bool passiveInit )
{
  passive = passiveInit;
  dirtySet();
}
bool ClientInfo::passiveGet()
{
  return passive;
}
// scheduler.cpp
// Copyright (c) 2004, Alio TV. All Rights Reserved.
#include "scheduler.h"
#include "stdio.h"
#include "sys/time.h"
#include "time.h"
#define RESOLUTION                  1000
```

```
#define SCHEDULER_ID    -1
#define SCHEDULER_CLIENT_CONTACT_TIMEOUT  ( 35 * 60 )
#define SCHEDULER_STRING_SIZE            1000
#define SCHEDULER_CUSTOMER_COUNT_MAX     10000
#define SCHEDULER_SPAWN_QUICK_INTERVAL   4
#define SCHEDULER_SPAWN_COUNT            100
#define SCHEDULER_CUSTOMER_LIST_SIZE     12
#define SCHEDULER_MEDIA_COUNT            34
#define SCHEDULER_MESSAGE_RETRIES        3
#define SCHEDULER_TRANSFER_SIZE          ( 2000000000LL / 10 )
#define SCHEDULER_TRANSFER_TIMEOUT       ( 3 * 60 )
#define SCHEDULER_TRANSFER_COUNT_MAX     50
#define SCHEDULER_CLIENT_DOWNLINK_DEFAULT 1000000
#define SCHEDULER_CLIENT_UPLINK_DEFAULT   128000
#define SCHEDULER_SERVER_DOWNLINK_DEFAULT 45000000
#define SCHEDULER_SERVER_UPLINK_DEFAULT   45000000
//#define SCHEDULER_SERVER_DOWNLINK_DEFAULT 5000000
//#define SCHEDULER_SERVER_UPLINK_DEFAULT   5000000
#define SCHEDULER_MEDIA_AUTHORIZATION_TIME  (10 * 60)
//#define SCHEDULER_MEDIA_AUTHORIZATION_TIME  (48 * 60 * 60)
#define SCHEDULER_WATCHABLE_MAX          12
#define SCHEDULER_MEDIA_COST             ( 4.5 )
Scheduler* theScheduler;
void* threadRunCall( void* )
{
  theScheduler->threadRunning();
  return NULL;
}
Scheduler::Scheduler( int init )
{
  uiInterface = NULL;
  running = false;
  theScheduler = this;
  ipPort = 0;
  customerSpawnLast = 0;
  customerCount = 0;
  p2pEnable = false;
  serverEnable = false;
  // Create the mutex
  pthread_mutex_init( &runningMutex, NULL );
  databaseInitialize( init );
  messageEngine = new MessageEngine( SCHEDULER_ID, "alio_scheduler" );
  srandom( timeGet() );
  transferPriority = false;
}
void Scheduler::uiInterfaceSet( SchedulerUIInterface* uiInterfaceInit )
{
  uiInterface = uiInterfaceInit;
}
void Scheduler::run()
```

```
{
  if ( !running )
  {
    Log::LT( &alioSession, 3, "Scheduler", "RUN", "Run command issued from UI" );
    uiInterface->logUpdate();
    running = true;
    printf( "  running with port %d\n", ipPort );
    messageEngine->listeningPortSet( ipPort );
    messageEngine->run();
    // Fire up the input thread
    pthread_create( &runningThread, NULL, threadRunCall, NULL );
  }
}
void Scheduler::stop()
{
  if ( running )
  {
    Log::LT( &alioSession, 3, "Scheduler", "STOP", "Stop command issued from UI" );
    uiInterface->logUpdate();
    messageEngine->stop();
    running = false;
  }
}
void Scheduler::p2pEnableSet( bool enable )
{
  p2pEnable = enable;
}
void Scheduler::serverEnableSet( bool enable )
{
  serverEnable = enable;
}
long long Scheduler::timeGet()
{
  return time( 0 );
}
void Scheduler::threadRunning(  )
{
  struct timeval timeStart;
  struct timeval timeEnd;
  struct timeval timeDifference;
  printf( "RUNNING\n" );
  running = true;
  while ( running )
  {
    gettimeofday( &timeStart, NULL );
    iterate( timeGet() );
    gettimeofday( &timeEnd, NULL );
    timeSubtract( &timeDifference, &timeStart, &timeEnd );
    long long us = timeDifference.tv_sec * 1000000 + timeDifference.tv_usec;
    // printf( "    Took %lldms\n",  us / 1000  );
```

```
    long long overrun = us - ( RESOLUTION * 1000 );
    if ( overrun <= 0 )
    {
      usleep( ( RESOLUTION * 1000 ) - us );
      // printf( "    Took %lldms\n",  us / 1000  );
    }
    else
      printf( "  Overran time by %lldms\n", overrun / 1000 );
    // seconds+= RESOLUTION/1000;
    uiInterface->timeUpdate();
  }
  printf( "STOPPED\n" );
}
void Scheduler::iterate( long long seconds )
{
  bool logRefresh = false;
  bool messageRefresh = false;
  bool clientInfoRefresh = false;
//  printf ( "Iterate %lld\n", seconds );
  // Do a bunch of stufff - each one in it's own transaction
    // initiate transfers - alternate high and low priority
  transfersDesign( transferPriority );
  transferPriority = !transferPriority;
  // Check on any wayward transfers
  transfersMonitor( seconds );
  // Check on any wayward messages
  messagesMonitor( seconds );
  alioSession.transactionStart();
  if ( customerCount < SCHEDULER_CUSTOMER_COUNT_MAX && seconds > customerSpawnLast +
SCHEDULER_SPAWN_QUICK_INTERVAL )
  {
    for ( int i = 0; i < SCHEDULER_SPAWN_COUNT; i++ )
      customerNewCreate( );
    customerSpawnLast = seconds;
  }
  alioSession.transactionEnd();
  alioSession.transactionStart();
  printf( "Scheduler: Messages " );
  Array< Message* > ms;
  messageEngine->messagesReceiveByReceiver( &alioSession, &ms, SCHEDULER_ID );
  printf("Count %d\n", ms.sizeGet() );
  if ( ms.sizeGet() > 0 )
  {
    //printf( "    %d Incoming Messages\n", ms.sizeGet() );
    for ( int i = 0; i < ms.sizeGet(); i++ )
    {
      Message* m = ms.elementGet( i );
      //printf( "      %d: %s\n", i, m->messageGet() );
      Log::L( &alioSession, 3, "Scheduler", "Message In", m->messageGet() );
      processMessage( m );
```

```
      m->deleteObject();
      logRefresh = true;
      messageRefresh = true;
    }
  }
  alioSession.transactionEnd();
  //printf( "  Check Status\n" );
  alioSession.transactionStart();
  Array< ClientInfo* > cis;
  ClientInfo::FindByContactTimeout( &cis, &alioSession, seconds );
  if ( cis.sizeGet() > 0 )
  {
    printf( "    %d Clients Timedout\n", cis.sizeGet() );
    for ( int i = 0; i < cis.sizeGet(); i++ )
    {
      ClientInfo* ci = cis.elementGet( i );
      printf( "      %d: ClientId %d : Timeout %lld\n", i, ci->idGet( ), ci->contactTimeoutGet() );
      if ( !ci->passiveGet() )
      {
        ci->contactTimeoutSet( seconds + SCHEDULER_CLIENT_CONTACT_TIMEOUT  );
        ci->connectedSet( false );
        Message* m = new Message( );
        m->toIdSet( ci->idGet() );
        m->fromIdSet( SCHEDULER_ID );
        m->messageSet( "contactPassive( );" );
        m->timestampSet( seconds );
        messageEngine->messageSend( &alioSession, m );
        Log::L( &alioSession, 3, "Scheduler", "Message Out", m->messageGet() );
        logRefresh = true;
        clientInfoRefresh = true;
      }
    }
  }
  alioSession.transactionEnd();
  // do a search for clients with under populated boxes - sort by how bad it is
/*
  if ( logRefresh )
    uiInterface->logUpdate();
  if ( messageRefresh )
    uiInterface->messageUpdate();
  if ( clientInfoRefresh )
    uiInterface->clientInfoUpdate();
*/
}
bool Scheduler::transfersDesign( bool topPriority )
{
  if ( !p2pEnable && !serverEnable )
  {
    return false;
  }
```

```
// Each one in a transaction?
alioSession.transactionStart();
Array< ClientInfo* > desparados;
if ( topPriority )
{
  printf( "Scheduler: Downloads High Priority " );
  ClientInfo::FindTopPriorityDownload( &desparados, &alioSession );
}
else
{
  printf( "Scheduler: Downloads Low Priority " );
  ClientInfo::FindNormalPriorityDownload( &desparados, &alioSession );
}
int count = desparados.sizeGet();
printf("Count %d\n", count );
if ( count > 0 )
{
  printf( "    %d Clients Ready For Media %s\n", count, ( topPriority ? "High Priority" : "Low Priority" ) );
  for ( int i = 0; i < desparados.sizeGet(); i++ )
  {
    ClientInfo* receiver = desparados.elementGet( i );
    printf( "      ClientId %d ", receiver->idGet( ) );
    // Find a media item
    int nextMediaItem = receiver->mediaItemForDownloadNextGet();
    if ( nextMediaItem != -1 )
    {
      printf( " Media Item %d ", nextMediaItem );
      // Find a donor
      Array< ClientInfo* > suppliers;
      if ( receiver->preloadingGet() )
        ClientInfo::FindServers( &suppliers, &alioSession );
      else
      {
        // ensure that if the receiver is a "passive" i.e. can not receive calls, can't chose a donor that is "passive" too
        ClientInfo::FindMediaSource( &suppliers, &alioSession, nextMediaItem, p2pEnable, serverEnable, receiver-
>passiveGet() );
      }
      if ( suppliers.sizeGet() > 0 )
      {
        ClientInfo *sender = suppliers.elementGet( 0 );
        // need to confirm that the sender has still got bandwidth
        // might have been commited in a previous loop
        printf( " Sender %d ", sender->idGet() );
        // Find the actual client media record for the receiver
        ClientMedia *cm = receiver->clientMediaGetFromMediaId( nextMediaItem );
        // Set the flag in Client Media to signal downloading
        cm->downloadingSet( cm->downloadingGet() + 1 );
        // Make a transfer record
        Transfer* transfer = new Transfer( &alioSession );
        transfer->toClientIdSet( receiver->idGet() );
```

```
        transfer->fromClientIdSet( sender->idGet() );
        transfer->mediaIdSet( nextMediaItem );
        transfer->transferByteStartSet( cm->sizeCurrentGet() );
        transfer->transferByteCurrentSet( cm->sizeCurrentGet() );
        transfer->toContactProcess( timeGet(), SCHEDULER_TRANSFER_TIMEOUT );
        transfer->fromContactProcess( timeGet(), SCHEDULER_TRANSFER_TIMEOUT );
        long long remainingSize = cm->sizeGet() - cm->sizeCurrentGet();
        long long transferSize = ( remainingSize < SCHEDULER_TRANSFER_SIZE ) ? remainingSize :
SCHEDULER_TRANSFER_SIZE;
        transfer->transferByteLengthSet( transferSize );
        // Send the transfer start message to the clients
        //   Arg 0 = "transferStart"
        //   Arg 1 = scheduler transfer id
        //   Arg 2 = remote id
        //   Arg 3 = remote ip address
        //   Arg 4 = remote transfer port
        //   Arg 5 = media id
        //   Arg 6 = active
        //   Arg 7 = sending
        //   Arg 8 = start byte
        //   Arg 9 = length
        char t[ SCHEDULER_STRING_SIZE ];
        Message* mr = new Message( );
        mr->toIdSet( receiver->idGet() );
        mr->fromIdSet( SCHEDULER_ID );
        mr->passiveSendSet( receiver->passiveGet() );
        snprintf( t, SCHEDULER_STRING_SIZE, "transferStart( %d, %d, %s, %d, %d, %d, %d, %lld, %lld )",
                transfer->idGet(), sender->idGet(), sender->ipAddressGet(), sender->ipTransferPortGet(),
nextMediaItem, receiver->passiveGet(), 0,
                transfer->transferByteStartGet(), transfer->transferByteLengthGet() );
        mr->messageSet( t );
        messageEngine->messageSend( &alioSession, mr );
        Message* ms = new Message( );
        ms->toIdSet( sender->idGet() );
        ms->fromIdSet( SCHEDULER_ID );
        ms->passiveSendSet( sender->passiveGet() );
        snprintf( t, SCHEDULER_STRING_SIZE, "transferStart( %d, %d, %s, %d, %d, %d, %d, %lld, %lld )",
                transfer->idGet(), receiver->idGet(), receiver->ipAddressGet(), receiver->ipTransferPortGet(),
nextMediaItem, !receiver->passiveGet(), 1,
                transfer->transferByteStartGet(), transfer->transferByteLengthGet() );
        ms->messageSet( t );
        messageEngine->messageSend( &alioSession, ms );
        // Get the likely download utilization
        int uplinkRateMax = sender->uplinkCapacityGet() - sender->uplinkCurrentGet();
        int downlinkRateMax = receiver->downlinkCapacityGet() - receiver->downlinkCurrentGet();
        int transferRate = ( uplinkRateMax < downlinkRateMax ) ? uplinkRateMax : downlinkRateMax;
        transfer->transferRateIdealSet( transferRate );
        // preloads don't load the server - see processMessageTransferComplete() & transferAbort( ) for the opposite
        if ( !( sender->serverGet() && receiver->preloadingGet() ) )
          sender->uplinkCurrentSet( sender->uplinkCurrentGet() + transferRate );
```

```
        receiver->downlinkCurrentSet( receiver->downlinkCurrentGet() + transferRate );
        // Other stuff to do?
       }
      }
      printf( "\n" );
    }
  }
  alioSession.transactionEnd();
  return  count < SCHEDULER_TRANSFER_COUNT_MAX;
}
void Scheduler::transfersMonitor( long long time  )
{
  alioSession.transactionStart();
  Array <Transfer*> transfers;
  Transfer::FindByContactTimeout( &transfers, &alioSession, time );
  int count = transfers.sizeGet();
  for ( int i = 0; i < count; i++ )
  {
    Transfer *transfer = transfers.elementGet( i );
    // clobber it
    transferAbort( transfer );
    transfer->deleteObject();
  }
  alioSession.transactionEnd();
}
void Scheduler::messagesMonitor( long long UNUSED time  )
{
  alioSession.transactionStart();
  Array <Message*> messages;
  Message::FindByRetriedOut( &messages, &alioSession, SCHEDULER_ID, SCHEDULER_MESSAGE_RETRIES );
  int count = messages.sizeGet();
  for ( int i = 0; i < count; i++ )
  {
    Message *message = messages.elementGet( i );
    Log::L( &alioSession, 3, "Scheduler", "Abandoning Message - too old", message->messageGet() );
    message->deleteObject();
  }
  alioSession.transactionEnd();
}
void Scheduler::transferAbort( Transfer* transfer )
{
  // specific action will depend on the state the different communicands find themselves in
  ClientInfo* sender = ClientInfo::Find( &alioSession, transfer->fromClientIdGet() );
  ClientInfo* receiver = ClientInfo::Find( &alioSession, transfer->toClientIdGet() );
  if ( sender )
  {
    // preloads don't load the server - see transferDesign( ) for the opposite
    if ( !( sender->serverGet() && receiver && receiver->preloadingGet() ) )
      sender->uplinkCurrentSet( sender->uplinkCurrentGet() - transfer->transferRateIdealGet() );
    if ( sender->connectedGet() )
```

```
    {
      transferAbortMessageSend( transfer->fromClientIdGet(), transfer->idGet() );
      sender->connectedSet( false );
    }
  }
  if ( receiver )
  {
    receiver->downlinkCurrentSet( receiver->downlinkCurrentGet() - transfer->transferRateIdealGet() );
    if ( receiver->connectedGet() )
    {
      transferAbortMessageSend( transfer->toClientIdGet(), transfer->idGet() );
      receiver->connectedSet( false );
    }
  }
  ClientMedia *cm = ClientMedia::FindByClientIdAndMediaId( &alioSession, transfer->toClientIdGet(), transfer -
>mediaIdGet() );
  if ( cm )
    cm->downloadingSet( cm->downloadingGet() - 1 );
}
void Scheduler::processMessage( Message* message )
{
  Array< char* > strings;
  message->messageParse( &strings );
  if ( strings.sizeGet() > 0 )
  {
    char* function = strings.elementGet( 0 );
    if ( strcmp( function, "clientNew" ) == 0 )
    {
      processMessageClientNew( &strings );
    }
    if ( strcmp( function, "ping" ) == 0 )
    {
      Log::L( &alioSession, 3, "Scheduler", "New Ping Message", message->messageGet() );
      processMessagePing( message->fromIdGet(), &strings );
    }
    if ( strcmp( function, "contact" ) == 0 )
    {
      Log::L( &alioSession, 3, "Scheduler", "New Contact Message", message->messageGet() );
      processMessageContact( message->fromIdGet(), &strings );
    }
    if ( strcmp( function, "clientMediaNew" ) == 0 )
    {
      Log::L( &alioSession, 3, "Scheduler", "New Client Media", message->messageGet() );
      processMessageClientMediaNew( message->fromIdGet(), &strings );
    }
    if ( strcmp( function, "clientMediaComplete" ) == 0 )
    {
      Log::L( &alioSession, 3, "Scheduler", "Client Media Complete", message->messageGet() );
      processMessageClientMediaComplete( message->fromIdGet(), &strings );
    }
```

```
if ( strcmp( function, "transferStarting" ) == 0 )
{
  Log::L( &alioSession, 3, "Scheduler", "Transfer Starting", message->messageGet() );
  processMessageTransferStarting( message->fromIdGet(), &strings );
}
if ( strcmp( function, "transferListening" ) == 0 )
{
  Log::L( &alioSession, 3, "Scheduler", "Transfer Listening", message->messageGet() );
  processMessageTransferListening( message->fromIdGet(), &strings );
}
if ( strcmp( function, "transferConnecting" ) == 0 )
{
  Log::L( &alioSession, 3, "Scheduler", "Transfer Connecting", message->messageGet() );
  processMessageTransferConnecting( message->fromIdGet(), &strings );
}
if ( strcmp( function, "transferTransferring" ) == 0 )
{
  Log::L( &alioSession, 3, "Scheduler", "Transfer Transfering", message->messageGet() );
  processMessageTransferTransferring( message->fromIdGet(), &strings );
}
if ( strcmp( function, "transferProgress" ) == 0 )
{
  Log::L( &alioSession, 3, "Scheduler", "Transfer Progress", message->messageGet() );
  processMessageTransferProgress( message->fromIdGet(), &strings );
}
if ( strcmp( function, "transferComplete" ) == 0 )
{
  Log::L( &alioSession, 3, "Scheduler", "Transfer Complete", message->messageGet() );
  processMessageTransferComplete( message->fromIdGet(), &strings );
}
if ( strcmp( function, "transferTimeout" ) == 0 )
{
  Log::L( &alioSession, 3, "Scheduler", "Transfer Timeout", message->messageGet() );
  processMessageTransferTimeout( message->fromIdGet(), &strings );
}
if ( strcmp( function, "transferError" ) == 0 )
{
  Log::L( &alioSession, 3, "Scheduler", "Transfer Error", message->messageGet() );
  processMessageTransferError( message->fromIdGet(), &strings );
}
if ( strcmp( function, "mediaAuthorizationRequest" ) == 0 )
{
  Log::L( &alioSession, 3, "Scheduler", "MediaAuthorizationRequest", message->messageGet() );
  processMessageMediaAuthorizationRequest( message->fromIdGet(), &strings );
}
if ( strcmp( function, "clientMediaMove" ) == 0 )
{
  Log::L( &alioSession, 3, "Scheduler", "ClientMediaMove", message->messageGet() );
  processMessageClientMediaMove( message->fromIdGet(), &strings );
}
```

```
    if ( strcmp( function, "clientMediaRemove" ) == 0 )
    {
      Log::L( &alioSession, 3, "Scheduler", "ClientMediaRemove", message->messageGet() );
      processMessageClientMediaRemove( message->fromIdGet(), &strings );
    }
    if ( strcmp( function, "clientMediaAdd" ) == 0 )
    {
      Log::L( &alioSession, 3, "Scheduler", "ClientMediaAdd", message->messageGet() );
      processMessageClientMediaAdd( message->fromIdGet(), &strings );
    }
  }
  for ( int i = 0; i < strings.sizeGet(); i++ )
    free( strings.elementGet( i ) );
}
void Scheduler::processMessageClientNew( Array< char* >* arguments )
{
  // Arg 0 = "clientNew"
  // Arg 1 = client id
  // Arg 2 = server?
  // Arg 3 = simulated?
  // Arg 4 = passive?
  // Arg 5 = ip address
  // Arg 6 = ip message port
  // Arg 7 = ip transfer port
  // Arg 8 = disk capacity
  int idInit;
  sscanf( arguments->elementGet( 1 ), "%d", &idInit );
  // Try to find the record for this client
  ClientInfo *clientInfo = ClientInfo::Find( &alioSession, idInit );
  if ( clientInfo == 0 )
    clientInfo = new ClientInfo( &alioSession, idInit );
  int serverInit;
  sscanf( arguments->elementGet( 2 ), "%d", &serverInit );
  clientInfo->serverSet( ( serverInit != 0 ) );
  int simulatedInit;
  sscanf( arguments->elementGet( 3 ), "%d", &simulatedInit );
  clientInfo->simulatedSet( ( simulatedInit != 0 ) );
  int passiveInit;
  sscanf( arguments->elementGet( 4 ), "%d", &passiveInit );
  clientInfo->passiveSet( ( passiveInit != 0 ) );
  clientInfo->ipAddressSet( arguments->elementGet( 5 ) );
  int ipMessagePortInit;
  sscanf( arguments->elementGet( 6 ), "%d", &ipMessagePortInit );
  clientInfo->ipMessagePortSet( ipMessagePortInit );
  int ipTransferPortInit;
  sscanf( arguments->elementGet( 7 ), "%d", &ipTransferPortInit );
  clientInfo->ipTransferPortSet( ipTransferPortInit );
  clientInfo->customerIdSet( -1 );
  long long storageCapacityInit;
  sscanf( arguments->elementGet( 8 ), "%lld", &storageCapacityInit );
```

```
  clientInfo->storageCapacitySet( storageCapacityInit );
  clientInfo->contactLastSet( timeGet() );
  clientInfo->contactTimeoutSet( timeGet() + SCHEDULER_CLIENT_CONTACT_TIMEOUT );
  printf( "New Client Created : %d \n", clientInfo->idGet() );
  // If it's a server, it doesn't get a customer id
  Customer* customer = 0;
  if ( !clientInfo->serverGet() )
  {
    Array< Customer* > customers;
    Customer::FindByClientNew( &customers, &alioSession );
    if ( customers.sizeGet() == 0 )
    {
      Log::L( &alioSession, 3, "Scheduler", "No New Customer", "Attempting to create a new customer on demand" );
      customerNewCreate( );
      Customer::FindByClientNew( &customers, &alioSession );
      if ( customers.sizeGet() == 0 )
      {
        Log::L( &alioSession, 2, "Scheduler", "Couldn't make a new Customer", "No customer available : ignoring" );
        return;
      }
    }
    customer = customers.elementGet( 0 );
    customer->clientNewSet( customer->clientNewGet() - 1 );
    clientInfo->customerIdSet( customer->idGet() );
    clientInfo->downlinkCapacitySet( SCHEDULER_CLIENT_DOWNLINK_DEFAULT );
    clientInfo->uplinkCapacitySet( SCHEDULER_CLIENT_UPLINK_DEFAULT );
  }
  else
  {
    clientInfo->downlinkCapacitySet( SCHEDULER_SERVER_DOWNLINK_DEFAULT );
    clientInfo->uplinkCapacitySet( SCHEDULER_SERVER_UPLINK_DEFAULT );
  }
  // Debugging...
  // clientInfo->commitDBObject();
  Message *clientConfirm = new Message( &alioSession );
  clientConfirm->fromIdSet( SCHEDULER_ID );
  clientConfirm->toIdSet( clientInfo->idGet() );
  clientConfirm->passiveSendSet( clientInfo->passiveGet() );
  char text[ 1000 ];
  snprintf( text, 1000, "clientNewConfirm( %d, %s, %d, %d )", clientInfo->idGet(), ( customer != 0 ) ? customer-
>nameGet() : "SERVER", clientInfo->uplinkCapacityGet(), clientInfo->downlinkCapacityGet() );
  clientConfirm->messageSet( text );
  // Send Media Items
  // No need to send items to the server - the simulator automagically does this
  if ( clientInfo->serverGet() || !clientInfo->simulatedGet() )
  {
    Array< MediaInfo* > mediaInfo;
    MediaInfo::FindByCatalog( &mediaInfo, &alioSession, 0 );
    for ( int i = 0; i < mediaInfo.sizeGet(); i++ )
    {
```

```
    MediaInfo *mi = mediaInfo.elementGet( i );
    // Send the info about the media item to the client
    Message *m = new Message( &alioSession );
    m->fromIdSet( SCHEDULER_ID );
    m->toIdSet( clientInfo->idGet() );
    m->passiveSendSet( clientInfo->passiveGet() );
    char s[ SCHEDULER_STRING_SIZE ];
    sprintf( s, "mediaInfo( %d, %d, \"%s\", \"%s\", \"%s\", \"%s\", %d, \"%s\", %lld )",
             mi->idGet(),
             mi->catalogIdGet(),
             mi->genreGet(),
             mi->titleGet(),
             mi->descriptionGet(),
             mi->directorGet(),
             mi->releaseYearGet(),
             mi->filenameGet(),
             mi->sizeGet() );
    m->messageSet( s );
  }
}
if ( !clientInfo->serverGet() )
{
  int clientMediaCount = 0;
  // Create Client Media Items and messages to the client for each one
  Array< CustomerMedia* > customerMedia;
  CustomerMedia::FindForNewClient( &customerMedia, &alioSession, customer->idGet() );
  int customerClientId = -1;
  for ( int i = 0; i < customerMedia.sizeGet(); i++ )
  {
    CustomerMedia *c = customerMedia.elementGet( i );
    if ( customerClientId == -1 )
      customerClientId = c->customerClientIdGet();
    // need to only do those customer media items that are for the same box
    if ( customerClientId == c->customerClientIdGet() )
    {
      clientMediaCount++;
      ClientMedia *cm = new ClientMedia( &alioSession );
      cm->clientIdSet( clientInfo->idGet() );
      cm->mediaIdSet( c->mediaIdGet() );
      cm->rankSet( c->rankGet() );
      MediaInfo* mi = MediaInfo::Find( &alioSession, c->mediaIdGet() );
      cm->sizeSet( mi->sizeGet() );
      bool complete = ( clientInfo->simulatedGet() || clientInfo->idGet() == 0 );
      cm->sizeCurrentSet( ( complete ) ? cm->sizeGet() : 0 );
      cm->completeSet( complete );
      cm->requestedSet( true );
      if ( complete )
      {
        clientInfo->storageCurrentSet( clientInfo->storageCurrentGet() + cm->sizeGet() );
        clientInfo->mediaAvailableCountSet( clientMediaCount );
```

```
        clientInfo->mediaUnwatchedCountSet( clientMediaCount );
      }
      Message *m = new Message( &alioSession );
      m->fromIdSet( SCHEDULER_ID );
      m->toIdSet( clientInfo->idGet() );
      m->passiveSendSet( clientInfo->passiveGet() );
      char s[ SCHEDULER_STRING_SIZE ];
      // For testing only, if a client is simulated (or a server) let's pretend that the media is loaded already
      sprintf( s, "mediaQueue( %d, %d, %d )", c->mediaIdGet(), c->rankGet(), complete  );
      m->messageSet( s );
     }
     clientInfo->mediaListSizeSet( clientMediaCount );
    }
  }
  clientInfo->preloadingSet( !clientInfo->simulatedGet() );
  Log::L( &alioSession, 3, "Scheduler", "New Client", text );
}
void Scheduler::processMessagePing( int fromId, Array< char* > UNUSED * arguments )
{
  // Arg 0 = "clientNew"
  // Arg 1 = ip address
  // Arg 2 = ip port
  // Look the client up
  ClientInfo* client = ClientInfo::Find( &alioSession, fromId );
  // check them!
  Message *clientConfirm = new Message( &alioSession );
  clientConfirm->fromIdSet( SCHEDULER_ID );
  clientConfirm->toIdSet( fromId );
  clientConfirm->messageSet( "pingBack( )" );
  if ( client != 0 )
    clientConfirm->passiveSendSet( client->passiveGet() );
  char s[ 10 ];
  sprintf( s, "ClientId:%d", fromId );
  Log::L( &alioSession, 3, "Scheduler", "Ping Message", s );
  printf( "Ping from %d\n", fromId );
}
void Scheduler::processMessageContact( int fromId, Array< char* > UNUSED * arguments )
{
  // Arg 0 = "contact"
  // Update the client info
  ClientInfo *clientInfo = ClientInfo::Find( &alioSession, fromId );
  if ( clientInfo != 0 )
  {
    clientInfo->contactTimeoutSet( timeGet() + SCHEDULER_CLIENT_CONTACT_TIMEOUT  );
    clientInfo->connectedSet( true );
    clientInfo->contactLastSet( timeGet() );
    Log::L( &alioSession, 3, "Scheduler", "Contact Message", "ClientId:%d", fromId );
  }
  else
  {
```

```
    Log::L( &alioSession, 3, "Scheduler", "Contact Message from Unknown", "ClientId:%d", fromId );
  }
}
void Scheduler::processMessageClientMediaNew( int fromId, Array< char* >* arguments )
{
  // Arg 0 = "clientMediaNew"
  // Arg 1 = mediaId
  // Arg 3 = requested
  // Arg 4 = rank
  ClientInfo *clientInfo = ClientInfo::Find( &alioSession, fromId );
  if ( clientInfo != 0 )
  {
    int  mediaId = -1;
    int  requestedInit = 0;
    bool requested = false;
    int  rank = -1;
    sscanf( arguments->elementGet( 1 ), "%d", &mediaId );
    sscanf( arguments->elementGet( 2 ), "%d", &requestedInit );
    requested = ( requestedInit != 0 );
    sscanf( arguments->elementGet( 3 ), "%d", &rank );
    Log::L( &alioSession, 3, "Scheduler", "New Client Media Message", "MediaId:%d, Requested:%d, Rank:%d",
mediaId, requested, rank );
    clientInfo->clientMediaNew( mediaId, requested, rank );
  }
}
void Scheduler::processMessageClientMediaComplete( int fromId, Array< char* >* arguments )
{
  // Arg 0 = "clientMediaNew"
  // Arg 1 = mediaId
  ClientInfo *clientInfo = ClientInfo::Find( &alioSession, fromId );
  if ( clientInfo != 0 )
  {
    int  mediaId = -1;
    sscanf( arguments->elementGet( 1 ), "%d", &mediaId );
    Log::L( &alioSession, 3, "Scheduler", "Client Media Complete", "MediaId:%d", mediaId );
    clientInfo->clientMediaComplete( mediaId );
  }
}
void Scheduler::processMessageTransferStarting( int fromId, Array< char* >* arguments )
{
  transferStateSet( fromId, Transfer::TRANSFER_STARTING, arguments );
}
void Scheduler::processMessageTransferListening( int fromId, Array< char* >* arguments )
{
  transferStateSet( fromId, Transfer::TRANSFER_LISTENING, arguments );
}
void Scheduler::processMessageTransferConnecting( int fromId, Array< char* >* arguments )
{
  transferStateSet( fromId, Transfer::TRANSFER_CONNECTING, arguments );
}
```

```
void Scheduler::processMessageTransferTransferring( int fromId, Array< char* >* arguments )
{
 transferStateSet( fromId, Transfer::TRANSFER_TRANSFERRING, arguments );
}
void Scheduler::processMessageTransferProgress( int fromId, Array< char* >* arguments )
{
 Transfer* transfer = transferStateSet( fromId, Transfer::TRANSFER_PROGRESS, arguments );
 if ( transfer )
 {
  //ClientInfo* receiver = ClientInfo::Find( &alioSession, transfer->toClientIdGet() );
  int localSends;
  long long bytes;
  long long seconds;
  sscanf( arguments->elementGet( 2 ), "%d", &localSends );
  sscanf( arguments->elementGet( 3 ), "%lld", &seconds );
  sscanf( arguments->elementGet( 4 ), "%lld", &bytes );
  // only bump up the count when the receiver says so
  if ( !localSends )
   transfer->transferByteCurrentSet( transfer->transferByteCurrentGet() + bytes );
 }
}
void Scheduler::processMessageTransferComplete( int fromId, Array< char* >* arguments )
{
 Transfer* transfer = transferStateSet( fromId, Transfer::TRANSFER_COMPLETE, arguments );
 if ( transfer )
 {
  if ( transfer->fromStateGet() == Transfer::TRANSFER_COMPLETE  &&
       transfer->toStateGet() == Transfer::TRANSFER_COMPLETE )
  {
   Log::L( &alioSession, 3, "Scheduler", "Transfer Complete", "%d - Cleaning up", transfer->idGet() );
   ClientInfo* sender = ClientInfo::Find( &alioSession, transfer->fromClientIdGet() );
   ClientInfo* receiver = ClientInfo::Find( &alioSession, transfer->toClientIdGet() );
   // preloads don't load the server - see transferDesign( ) for the opposite
   if ( !( sender->serverGet() && receiver->preloadingGet() ) )
    sender->uplinkCurrentSet( sender->uplinkCurrentGet() - transfer->transferRateIdealGet() );
   receiver->downlinkCurrentSet( receiver->downlinkCurrentGet() - transfer->transferRateIdealGet() );
   // find the clientMedia for the sender
   ClientMedia* clientMedia = receiver->clientMediaGetFromMediaId( transfer->mediaIdGet() );
   if ( clientMedia != 0 )
   {
    clientMedia->downloadingSet( clientMedia->downloadingGet() - 1 );
    clientMedia->sizeCurrentSet( clientMedia->sizeCurrentGet() + transfer->transferByteLengthGet() );
    if ( clientMedia->sizeCurrentGet() == clientMedia->sizeGet() )
    {
     clientMedia->completeSet( true );
     // Inform the client that the Media is officially complete
     Message* m = new Message( );
     m->toIdSet( receiver->idGet() );
     m->fromIdSet( SCHEDULER_ID );
     m->passiveSendSet( receiver->passiveGet() );
```

```
        char s[ SCHEDULER_STRING_SIZE ];
        sprintf( s, "mediaComplete( %d );", transfer->mediaIdGet() );
        m->messageSet( s );
        m->timestampSet( timeGet() );
        messageEngine->messageSend( &alioSession, m );
        // make sure the receiver's media count is right
        receiver->mediaProcess( timeGet() );
        // Make sure the client sorts out
        receiver->downlinkLastSet( timeGet() );
        // Check to see if the client is done preloading
        if ( receiver->preloadingGet() )
        {
          if ( receiver->mediaAvailableCountGet() >= SCHEDULER_WATCHABLE_MAX )
            receiver->preloadingSet( false );
        }
        Log::L( &alioSession, 3, "Scheduler", "Media Complete Message", m->messageGet() );
      }
    }
    // Update the space on disk
    receiver->storageCurrentSet( receiver->storageCurrentGet() + transfer->transferByteLengthGet() );
    // Now get all the media counters, etc. up to date
    receiver->mediaProcess( timeGet() );
    // Now dump the transfer record
    transfer->deleteObject();
  }
 }
}
void Scheduler::processMessageTransferTimeout( int UNUSED fromId, Array< char* > UNUSED * arguments )
{
 //transferStateSet( fromId, Transfer::TRANSFER_ );
}
/**  Cancel the transfer for the other party too
 **/
void Scheduler::processMessageTransferError( int UNUSED fromId, Array< char* > * arguments )
{
 int  transferId = -1;
 sscanf( arguments->elementGet( 1 ), "%d", &transferId );
 Transfer* transfer = Transfer::Find( &alioSession, transferId );
 if ( transfer )
 {
  // put them both offline for a bit
  ClientInfo* ci;
  ci = ClientInfo::Find( &alioSession, transfer->toClientIdGet() );
  if ( ci )
    ci->connectedSet( false );
  ci = ClientInfo::Find( &alioSession, transfer->fromClientIdGet() );
  if ( ci )
    ci->connectedSet( false );
  transferAbort( transfer );
  transfer->deleteObject();
```

```
  }
}
/**  Media Authorization Request
 **/
void Scheduler::processMessageMediaAuthorizationRequest( int fromId, Array< char* > * arguments )
{
  // Arg 0: mediaAuthorizationRequest
  // Arg 1: mediaId
  int  mediaInfoId = -1;
  sscanf( arguments->elementGet( 1 ), "%d", &mediaInfoId );
  ClientInfo* clientInfo = ClientInfo::Find( &alioSession, fromId );
  if ( clientInfo )
  {
    ClientMedia* clientMedia = clientInfo->clientMediaGetFromMediaId( mediaInfoId );
    if ( clientMedia )
    {
      // Debit client account
      Customer* customer = Customer::Find( &alioSession, clientInfo->customerIdGet() );
      if ( customer )
      {
        customer->accountSet( customer->accountGet() + SCHEDULER_MEDIA_COST );
      }
      // Keep a record here
      clientMedia->authorize( timeGet(), SCHEDULER_MEDIA_AUTHORIZATION_TIME );
      // Authorize the movie
      Message* m = new Message( );
      m->toIdSet( fromId );
      m->fromIdSet( SCHEDULER_ID );
      m->passiveSendSet( clientInfo->passiveGet() );
      char s[ SCHEDULER_STRING_SIZE ];
      sprintf( s, "mediaAuthorize( %d );", mediaInfoId );
      m->messageSet( s );
      m->timestampSet( timeGet() );
      messageEngine->messageSend( &alioSession, m );
    }
  }
}
void Scheduler::processMessageClientMediaMove( int fromId, Array< char* >* arguments )
{
  // Arg 0: mediaAuthorizationRequest
  // Arg 1: rank old
  // Arg 2: rank new
  int  rankOld = -1;
  sscanf( arguments->elementGet( 1 ), "%d", &rankOld );
  int  rankNew = -1;
  sscanf( arguments->elementGet( 2 ), "%d", &rankNew );
  mediaItemMove( fromId, rankOld, rankNew );
}
void Scheduler::processMessageClientMediaAdd( int fromId, Array< char* >* arguments )
{
```

```
  // Arg 0: mediaAuthorizationRequest
  // Arg 1: rank
  // Arg 2: mediaId
  int  rank = -1;
  sscanf( arguments->elementGet( 1 ), "%d", &rank );
  int  mediaId = -1;
  sscanf( arguments->elementGet( 2 ), "%d", &mediaId );
  mediaItemAdd( fromId, rank, mediaId );
}
void Scheduler::processMessageClientMediaRemove( int fromId, Array< char* >* arguments )
{
  // Arg 0: mediaAuthorizationRequest
  // Arg 1: rank
  int  rank = -1;
  sscanf( arguments->elementGet( 1 ), "%d", &rank );
  mediaItemRemove( fromId, rank );
}
void Scheduler::mediaItemRemove( int clientId, int rank )
{
  if ( clientId != -1 && rank != -1 )
  {
    ClientInfo* ci = ClientInfo::Find( &alioSession, clientId );
    if ( ci )
      ci->clientMediaRemove( rank );
    Log::L( &alioSession, 3, "Scheduler", "mediaItemRemove", "Succeeded Client %d Rank %d", clientId, rank );
  }
  else
    Log::L( &alioSession, 2, "Scheduler", "mediaItemRemove", "Failed Client %d Rank %d", clientId, rank );
}
void Scheduler::mediaItemMove( int clientId, int rankOld, int rankNew )
{
  if ( clientId != -1 && rankOld != -1 && rankNew != -1 )
  {
    ClientInfo* ci = ClientInfo::Find( &alioSession, clientId );
    if ( ci )
      ci->clientMediaMove( rankOld, rankNew );
    Log::L( &alioSession, 3, "Scheduler", "mediaItemMove", "Succeeded Client %d RankOld %d RankNew %d",
clientId, rankOld, rankNew );
  }
  else
    Log::L( &alioSession, 2, "Scheduler", "mediaItemMove", "Failed Client %d RankOld %d RankNew %d", clientId,
rankOld, rankNew );
}
void Scheduler::mediaItemAdd( int clientId, int rank, int mediaId )
{
  if ( clientId != -1 && rank != -1 && mediaId != -1 )
  {
    ClientInfo* ci = ClientInfo::Find( &alioSession, clientId );
    if ( ci )
      ci->clientMediaAdd( rank, mediaId );
```

```
    Log::L( &alioSession, 3, "Scheduler", "mediaItemAdd", "Succeeded Client %d Rank %d Media %d", clientId, rank,
mediaId );
  }
  else
    Log::L( &alioSession, 2, "Scheduler", "mediaItemAdd", "Failed Client %d Rank %d Media %d", clientId, rank,
mediaId );
}
Transfer* Scheduler::transferStateSet( int fromId, Transfer::State state, Array< char* >* arguments )
{
  Transfer* transfer = 0;
  int  transferId = -1;
  sscanf( arguments->elementGet( 1 ), "%d", &transferId );
  transfer = Transfer::Find( &alioSession, transferId );
  if ( transfer )
  {
    if ( fromId == transfer->fromClientIdGet() )
    {
      transfer->fromStateSet( state );
      transfer->fromContactProcess( timeGet(), SCHEDULER_TRANSFER_TIMEOUT );
    }
    else
    {
      transfer->toStateSet( state );
      transfer->toContactLastSet( timeGet() );
      transfer->toContactProcess( timeGet(), SCHEDULER_TRANSFER_TIMEOUT );
    }
    Log::L( &alioSession, 3, "Scheduler", "Transfer State Change", "TransferId:%d Client %d State %d", transferId,
fromId, (int)state );
  }
  else
  {
    Log::L( &alioSession, 2, "Scheduler", "Transfer State Change Error", "Cant find TransferId:%d Client %d State %d",
transferId, fromId, (int)state );
    transferAbortMessageSend( fromId, transferId );
  }
  return transfer;
}
void Scheduler::transferAbortMessageSend( int clientId, int transferId )
{
  ClientInfo* ci = ClientInfo::Find( &alioSession, clientId );
  Message* m = new Message( );
  m->toIdSet( clientId );
  m->fromIdSet( SCHEDULER_ID );
  if ( ci != 0)
    m->passiveSendSet( ci->passiveGet() );
  char s[ SCHEDULER_STRING_SIZE ];
  sprintf( s, "transferAbort( %d );", transferId );
  m->messageSet( s );
  m->timestampSet( timeGet() );
  messageEngine->messageSend( &alioSession, m );
```

```
}
void Scheduler::customerNewCreate( )
{
  Customer *customer = new Customer( &alioSession );
  char n[ 100 ];
  if ( ( customer->idGet() % 2 ) == 0 )
    sprintf( n, "Mr %ld", random() % 10000 );
  else
    sprintf( n, "Ms %ld", random() % 10000 );
  customer->nameSet( n );
  customer->clientNewSet( 1 );
  int ms[ SCHEDULER_MEDIA_COUNT ];
  for ( int i = 0; i < SCHEDULER_MEDIA_COUNT; i++ )
  {
    ms[ i ] = i;
  }
  for ( int i = 0; i < SCHEDULER_MEDIA_COUNT; i++ )
  {
    int a = random() % SCHEDULER_MEDIA_COUNT;
    int b = random() % SCHEDULER_MEDIA_COUNT;
    int t;
    t = ms[ a ];
    ms[ a ] = ms[ b ];
    ms[ b ] = t;
  }
  CustomerMedia* customerMedia[ SCHEDULER_CUSTOMER_LIST_SIZE ];
  for ( int i = 0; i < SCHEDULER_CUSTOMER_LIST_SIZE; i++ )
  {
    customerMedia[ i ] = new CustomerMedia( &alioSession );
    customerMedia[ i ]->customerIdSet( customer->idGet() );
    customerMedia[ i ]->mediaIdSet( ms[ i ] );
    customerMedia[ i ]->rankSet( i );
  }
  customerCount++;
}
void Scheduler::databaseInitialize( int init )
{
  AlioError status = alioSession.connect( "localhost", "alio_scheduler", "test", 0 );
  if ( status != ALIO_OK )
  {
    printf( "  Couldn't connect to the DB %d\n", status );
    return;
  }
  if ( init )
  {
    MediaInfo::TableDrop( &alioSession );
    Log::TableDrop( &alioSession );
    Message::TableDrop( &alioSession );
    ClientInfo::TableDrop( &alioSession );
    ClientMedia::TableDrop( &alioSession );
```

```
    Transfer::TableDrop( &alioSession );
    Customer::TableDrop( &alioSession );
    CustomerMedia::TableDrop( &alioSession );
  }
  MediaInfo::TableConfirm( &alioSession );
  ClientInfo::TableConfirm( &alioSession );
  ClientMedia::TableConfirm( &alioSession );
  Transfer::TableConfirm( &alioSession );
  Message::TableConfirm( &alioSession );
  Log::TableConfirm( &alioSession );
  Customer::TableConfirm( &alioSession );
  CustomerMedia::TableConfirm( &alioSession );
  if ( init )
  {
    MediaInfo::SamplesCreate( &alioSession, 0, SCHEDULER_MEDIA_COUNT );
//    CustomerMedia::SamplesCreate( &alioSession, 5, 5 );
//    Customer::SamplesCreate( &alioSession, 0, 5 );
  }
}
void Scheduler::timeSubtract( struct timeval* diff, struct timeval* start, struct timeval* end )
{
  diff->tv_sec = end->tv_sec - start->tv_sec;
  diff->tv_usec = end->tv_usec - start->tv_usec;
  if (diff->tv_usec < 0)
  {
    --diff->tv_sec;
    diff->tv_usec += 1000000;
  }
}
// transfer_engine.cpp
// Copyright (c) 2004, Alio TV. All Rights Reserved.
#include <transfer_engine.h>
#include "stdio.h"
#include "time.h"
#include "sys/time.h"
#include <log.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>  // sockets
#include <netinet/in.h>  // sockets
#include <arpa/inet.h>  // sockets
#include <netdb.h>
#include <unistd.h>
#include <stdlib.h>   // malloc
#include <fcntl.h>
#include <time.h>
#include <errno.h>              // for CONREFUSED
#include <client_info.h>
#include <alio.h>
#define RESOLUTION                    1000
```

```
#define PATH_LENGTH                 100
#define TRANSFER_PROGRESS_THRESHOLD     10000000LL
TransferEngine* theTransferEngine;
static void* threadRunCall( void* transferRecordVP )
{
  TransferRecord* transferRecord = (TransferRecord*)transferRecordVP;
  transferRecord->transferEngine->threadRunning( transferRecord );
  return NULL;
}
TransferRecord::TransferRecord( TransferEngine* transferEngineInit,
                                int schedulerTransferIdInit,
                                int localIdInit,
                                int localTransferPortInit,
                                int localClientMediaIdInit,
                                int remoteIdInit,
                                char* remoteIPAddressInit,
                                int remoteTransferPortInit,
                                bool localActiveInit, bool localSendsInit,
                                char* localPathInit,
                                long long startInit,
                                long long lengthInit,
                                bool virtualTransferInit )
{
  transferEngine = transferEngineInit;
  schedulerTransferId = schedulerTransferIdInit,
  localId = localIdInit,
  localTransferPort = localTransferPortInit,
  localClientMediaId = localClientMediaIdInit;
  remoteId = remoteIdInit;
  if ( remoteIPAddressInit != 0 )
    remoteIPAddress = strdup( remoteIPAddressInit );
  else
    remoteIPAddress = 0;
  if ( localPathInit != 0 )
    localPath = strdup( localPathInit );
  else
    localPath = 0;
  remoteTransferPort = remoteTransferPortInit;
  localActive = localActiveInit;
  localSends = localSendsInit;
  start = startInit;
  length = lengthInit;
  current = 0;
  virtualTransfer = virtualTransferInit;
}
TransferRecord::~TransferRecord( )
{
  if ( remoteIPAddress != 0 )
    free( remoteIPAddress );
  if ( localPath != 0 )
```

```
    free( localPath );
}
TransferEngine::TransferEngine( int idInit, const char UNUSED *databaseName )
{
  id = idInit;
  transferEngineCallbackInterface = 0;
  pathBase = strdup( "." );
  // databaseConnectionInitialize( databaseName );
}
void TransferEngine::pathBaseSet( const char* pathBaseInit )
{
  if ( pathBase != 0 )
    free( pathBase );
  pathBase = strdup( pathBaseInit );
}
void TransferEngine::transferStart( int schedulerTransferId,
                                    int localId,
                                    int localTransferPort,
                                    int localClientMediaId,
                                    int remoteId,
                                    char* remoteIPAddress,
                                    int remoteTransferPort,
                                    bool localActive, bool localSends,
                                    char* localPath,
                                    long long start,
                                    long long length,
                                    bool virtualTransfer )
{
  char p[ PATH_LENGTH ];
  sprintf( p, "%s/%s", pathBase, localPath );
  TransferRecord* transferRecord = new TransferRecord( this, schedulerTransferId,
                                                       localId,
                                                       localTransferPort,
                                                       localClientMediaId,
                                                       remoteId,
                                                       remoteIPAddress,
                                                       remoteTransferPort,
                                                       localActive, localSends,
                                                       p, start, length,
                                                       virtualTransfer );
  pthread_t transferThread;
  if ( pthread_create( &transferThread, NULL, threadRunCall, transferRecord ) )
  {
    printf( "THREAD CREATE FAILED\n" );
    delete transferRecord;
  }
  else
  {
    // this is risky
    transferRecord->thread = transferThread;
```

```
    pthread_detach( transferThread );
  }
}
void TransferEngine::transferCancel( int UNUSED schedulerTransferId )
{
}
void TransferEngine::threadRunning( TransferRecord* transferRecord )
{
  long long total_bytes_remaining;
  int mySocket;                 // TCP/IP stuff
  int acceptSocket;
  int transferSocket;    // which one (my or accept) to use for the transfer
  struct sockaddr_in socketInternetAddress;
  int fd;                       // fd for the file being transferred
  char *buffer;                 // read or write buffer
  int retval;
  struct stat statStruct;         // the reader needs to know the size of the file
  int retry;
  pthread_setcancelstate( PTHREAD_CANCEL_ENABLE, NULL );
  pthread_setcanceltype( PTHREAD_CANCEL_DEFERRED, NULL );
  printf( "TE TRANSFER START %s %lld %lld\n", transferRecord->localPath, transferRecord->start, transferRecord-
>length  );
  if( transferRecord->virtualTransfer )
  {
    sleep( 5 );
    if ( transferEngineCallbackInterface != 0 )
      transferEngineCallbackInterface->transferStarting( transferRecord->localId, transferRecord->schedulerTransferId
);
    sleep( 5 );
    if ( transferEngineCallbackInterface != 0 )
    {
      if ( transferRecord->localActive )
        transferEngineCallbackInterface->transferConnecting( transferRecord->localId, transferRecord-
>schedulerTransferId );
      else
        transferEngineCallbackInterface->transferListening( transferRecord->localId, transferRecord->schedulerTransferId
);
    }
    sleep( 5 );
    if ( transferEngineCallbackInterface != 0 )
      transferEngineCallbackInterface->transferTransferring( transferRecord->localId, transferRecord-
>schedulerTransferId );
    long long time = 0;
    //long long unit = 1966080;
    long long unit = 10000000;
    for ( long long i = transferRecord->start; i < transferRecord->start + transferRecord->length; i += unit )
    {
      sleep ( 5 );
      long long x;
      long long remaining;
```

```
      remaining = transferRecord->length - transferRecord->current;
      x = ( remaining < unit ) ? remaining : unit;
      transferRecord->current += x;
      time += 10;
      if ( transferEngineCallbackInterface != 0 )
      {
       // must send a PROGRESS message right before the transferComplete message is sent
       transferEngineCallbackInterface->transferProgress( transferRecord->localId, transferRecord-
>schedulerTransferId,
                                                          transferRecord->localClientMediaId, transferRecord->localSends, x, time );
      }
    }
    if ( transferEngineCallbackInterface != 0 )
      transferEngineCallbackInterface->transferComplete( transferRecord->localId, transferRecord-
>schedulerTransferId,
                                                        transferRecord->localClientMediaId, transferRecord->localSends );
    printf( "TE TRANSFER DONE\n" );
    // clean up
    delete transferRecord;
    return;
  } // if( transferRecord->virtualTransfer )
  // initialize stuff
  mySocket       = 0;
  acceptSocket   = 0;
  transferSocket = 0;
  fd             = 0;
  // prepare what we can of the socket address. the address and port depend on whether we're active or passive.
  bzero( ( char* ) &socketInternetAddress, sizeof( socketInternetAddress ) );
  socketInternetAddress.sin_family = AF_INET;
  // allocate the buffer
  if ( ! (buffer = ( char* ) malloc(MAX_BUF_SIZE) ) )
  {
    perror("malloc");
    if ( transferEngineCallbackInterface != 0 )
      transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
    if( mySocket     ) close( mySocket     );
    if( acceptSocket ) close( acceptSocket );
    if( fd           ) close( fd           );
    if( buffer       )  free( buffer       );
    delete transferRecord;
    return;
  }
  // open the file. if sender, open for reading
  if ( transferRecord->localSends )
  {
    fd = open(transferRecord->localPath, O_RDONLY );
    if( fd == -1 )
    {
      perror( "open");
      if ( transferEngineCallbackInterface != 0 )
```

```
    transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
   if( mySocket     ) close( mySocket     );
   if( acceptSocket ) close( acceptSocket );
   if( fd           ) close( fd           );
   if( buffer       )  free( buffer       );
   delete transferRecord;
   return;
  }
  // get the size of the file
  retval = fstat( fd, &statStruct);
  if( retval )
  {
   perror( "fstat");
   if ( transferEngineCallbackInterface != 0 )
    transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
   if( mySocket     ) close( mySocket     );
   if( acceptSocket ) close( acceptSocket );
   if( fd           ) close( fd           );
   if( buffer       )  free( buffer       );
   delete transferRecord;
   return;
  }
  printf( "file size %lld bytes, to transfer %lld, offset %lld\n",
        (long long) statStruct.st_size, transferRecord->length , transferRecord->start );
  // if the offset + the requested size > file size, complain
  if( transferRecord->length + transferRecord->start > ( long long )statStruct.st_size)
  {
   printf("file isn't big enough for the requested transfer\n");
   if ( transferEngineCallbackInterface != 0 )
    transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
   if( mySocket     ) close( mySocket     );
   if( acceptSocket ) close( acceptSocket );
   if( fd           ) close( fd           );
   if( buffer       )  free( buffer       );
   delete transferRecord;
   return;
  }
 }
 else
 { // we are the receiver, so open the file for writing, creating if necessary
  fd = open( transferRecord->localPath, O_WRONLY | O_CREAT, FILE_CREATE_MODE );
  if( fd == -1 )
  {
   perror( "open");
   if ( transferEngineCallbackInterface != 0 )
    transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
   if( mySocket     ) close( mySocket     );
   if( acceptSocket ) close( acceptSocket );
   if( fd           ) close( fd           );
   if( buffer       )  free( buffer       );
```

```
      delete transferRecord;
      return;
    }
  }
  // seek into the file. same for both sender and receiver
  retval = lseek( fd, transferRecord->start, SEEK_SET);
  if( retval == -1 )
  {
    perror( "lseek");
    if ( transferEngineCallbackInterface != 0 )
      transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
    if( mySocket     ) close( mySocket     );
    if( acceptSocket ) close( acceptSocket );
    if( fd           ) close( fd           );
    if( buffer       )  free( buffer       );
    delete transferRecord;
    return;
  }
  // open a socket; if we're the active partner, this will be our transfer socket.
  // on the other hand, if we're the passive partner, we'll use this socket to
  // listen for connections.
  if( ( mySocket = socket( AF_INET, SOCK_STREAM, 0 ) ) == -1 )
  {
    perror("socket");
    if ( transferEngineCallbackInterface != 0 )
      transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
    if( mySocket     ) close( mySocket     );
    if( acceptSocket ) close( acceptSocket );
    if( fd           ) close( fd           );
    if( buffer       )  free( buffer       );
    delete transferRecord;
    return;
  }
  // set up socket and make connection, depending on whether
  // we're the active or the passive party
  if ( transferRecord->localActive )
  {
    printf(" active partner - 1 sec\n" );
    sleep( 10 );
    printf(" active partner - calling %s\n", transferRecord->remoteIPAddress );
    // the socket address will be the remote address
    inet_pton(AF_INET, transferRecord->remoteIPAddress, &socketInternetAddress.sin_addr);
    socketInternetAddress.sin_port = htons( transferRecord->remoteTransferPort );
    if ( transferEngineCallbackInterface != 0 )
     transferEngineCallbackInterface->transferConnecting( transferRecord->localId, transferRecord-
>schedulerTransferId );
    retry = CONNECT_RETRIES_MAX;
    while( retry-- )
    {
      retval = connect( mySocket,
```

```
        (struct sockaddr *)&socketInternetAddress,
        sizeof(socketInternetAddress)) ;
    if ( retval == 0)
      break;
    if ( retval == -1)
    {
     if( ECONNREFUSED == errno )
     {
       printf("connection refused - will retry\n");
     }
     else
       perror("connect");
      if (CONNECT_RETRY_SLEEP_SECONDS ) sleep( CONNECT_RETRY_SLEEP_SECONDS );
       continue;
    }
  }
  // we're either connected or we've exhausted the retries
  if( retval )
  {
    printf ("connect: exhausted retries\n");
    if ( transferEngineCallbackInterface != 0 )
      transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
    if( mySocket     ) close( mySocket     );
    if( acceptSocket ) close( acceptSocket );
    if( fd           ) close( fd           );
    if( buffer       )  free( buffer       );
    delete transferRecord;
    return;
  }
  printf("active partner, connected\n");
  // note that this socket will be used for the transfer
  transferSocket = mySocket;
}
else // passive
{
  printf( "passive partner, calling accept\n" );
  socketInternetAddress.sin_addr.s_addr = INADDR_ANY;
  socketInternetAddress.sin_port = htons( transferRecord->localTransferPort );
  // allow reuse of the socket address to avoid bind failures
  int yes=1; // reuse the socket
  retval = setsockopt ( mySocket,
                        SOL_SOCKET,
   SO_REUSEADDR,
                        ( char* ) &yes,
   sizeof( yes ) );
  if( retval )
  {
    perror( "setsockopt" );
    if ( transferEngineCallbackInterface != 0 )
     transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
```

```
    if( mySocket     ) close( mySocket     );
    if( acceptSocket ) close( acceptSocket );
    if( fd           ) close( fd           );
    if( buffer       )  free( buffer       );
    delete transferRecord;
    return;
   }
   printf( "passive partner, binding\n" );
   // bind the socket to the address
   retry = BIND_RETRIES_MAX;
   while(retry--)
   {
    if( ( retval = bind( mySocket,
                         (struct sockaddr *) &socketInternetAddress,
                         sizeof( socketInternetAddress ) ) ) )
    {
     perror( "bind" );
     if (CONNECT_RETRY_SLEEP_SECONDS ) sleep( CONNECT_RETRY_SLEEP_SECONDS );
     continue;
    }
    break;
   }
   // we're either bound or we've exhausted the retries
   if( retval )
   {
    printf ("bind: exhausted retries\n");
    if ( transferEngineCallbackInterface != 0 )
      transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
    if( mySocket     ) close( mySocket     );
    if( acceptSocket ) close( acceptSocket );
    if( fd           ) close( fd           );
    if( buffer       )  free( buffer       );
    delete transferRecord;
    return;
   }
   if ( transferEngineCallbackInterface != 0 )
    transferEngineCallbackInterface->transferListening( transferRecord->localId, transferRecord->schedulerTransferId
);
   printf( "passive partner, listening\n" );
   if( listen( mySocket, 1 ) )
   {
    perror (" listen" );
    if ( transferEngineCallbackInterface != 0 )
      transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
    if( mySocket     ) close( mySocket     );
    if( acceptSocket ) close( acceptSocket );
    if( fd           ) close( fd           );
    if( buffer       )  free( buffer       );
    delete transferRecord;
    return;
```

```
  }
  struct timeval timeout;
  timeout.tv_sec = ACCEPT_TIMEOUT_SECONDS;
  timeout.tv_usec = ACCEPT_TIMEOUT_MILLISECONDS;
  printf( "passive partner, accepting\n" );
  acceptSocket = acceptWithTimeout(
                mySocket,
    &socketInternetAddress,
                timeout);
  if ( acceptSocket == -1 )
  {
    perror (" accept" );
    if ( transferEngineCallbackInterface != 0 )
     transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
    if( mySocket     ) close( mySocket     );
    if( acceptSocket ) close( acceptSocket );
    if( fd           ) close( fd           );
    if( buffer       )  free( buffer       );
    delete transferRecord;
    return;
  }
  printf( "client %s has connected\n", inet_ntoa( socketInternetAddress.sin_addr ) );
  transferSocket = acceptSocket;
  // close the listen socket, now the accept one is running
  if( mySocket )
  {
    close( mySocket );
    mySocket = 0;
  }
 }
 // start the transfer
 total_bytes_remaining = transferRecord->length;
 printf("have to transfer %lld bytes\n", total_bytes_remaining );
 if ( transferEngineCallbackInterface != 0 )
  transferEngineCallbackInterface->transferStarting( transferRecord->localId, transferRecord->schedulerTransferId );
 // make note of the starting time
 long long elapsedTime = 0;
 time_t timeStarted;
 time_t timeNow;
 if( -1 == time( &timeStarted ) )
 {
  perror("time");
  if ( transferEngineCallbackInterface != 0 )
   transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
 } // continue anyway, but elapsed time might be bogus
 if ( transferEngineCallbackInterface != 0 )
  transferEngineCallbackInterface->transferTransferring( transferRecord->localId, transferRecord-
>schedulerTransferId );
 long long transfered = 0;
 do
```

```
  {
    if ( transferRecord->localSends )
    { // sender
      int bytes_to_read_this_read;
      int bytes_read_this_read;
      int bytes_sent_this_send;
      int bytes_sent_this_read;
      memset(buffer, 0, MAX_BUF_SIZE);
      // how many bytes to ask for?
      bytes_to_read_this_read = ( total_bytes_remaining < MAX_BUF_SIZE )
                    ? total_bytes_remaining
        : MAX_BUF_SIZE ;
      bytes_read_this_read = read( fd, buffer, bytes_to_read_this_read );
      if( -1 == bytes_read_this_read )
      {
        perror( "read");
        if ( transferEngineCallbackInterface != 0 )
         transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
       if( mySocket     ) close( mySocket     );
       if( acceptSocket ) close( acceptSocket );
       if( fd           ) close( fd           );
       if( buffer       )  free( buffer       );
       delete transferRecord;
       return;
      }
      // printf( "remaining %lld bytes, read %d\n", total_bytes_remaining , bytes_read_this_read );
      if( 0 == bytes_read_this_read )
      {
        printf("end of file\n");
       if( total_bytes_remaining )
       {
         printf("ERROR: reached EOF but still have bytes to send\n");
         if ( transferEngineCallbackInterface != 0 )
           transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId
);
         if( mySocket     ) close( mySocket     );
         if( acceptSocket ) close( acceptSocket );
         if( fd           ) close( fd           );
         if( buffer       )  free( buffer       );
         delete transferRecord;
         return;
       }
       else
       {
         printf("LOGIC ERROR: still in loop but no bytes remaining\n");
         if ( transferEngineCallbackInterface != 0 )
           transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId
);
         if( mySocket     ) close( mySocket     );
         if( acceptSocket ) close( acceptSocket );
```

```
        if( fd          ) close( fd          );
        if( buffer      )  free( buffer      );
        delete transferRecord;
        return;
       }
      }
      if( bytes_to_read_this_read != bytes_read_this_read )
      {
       printf("read less than the buffer size: odd, but not an error\n");
      }
      bytes_sent_this_read =  0;
      do
      {
        bytes_sent_this_send = send( transferSocket,
                                     buffer + bytes_sent_this_read,
                                     bytes_read_this_read - bytes_sent_this_read,
                                    0);
       if( -1 == bytes_sent_this_send )
       {
        perror( "send" );
        if ( transferEngineCallbackInterface != 0 )
          transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId
);
        if( mySocket     ) close( mySocket     ); //TODO do this on every return
        if( acceptSocket ) close( acceptSocket );
        if( fd          ) close( fd          );
        if( buffer      )  free( buffer      );
        delete transferRecord;
        return;
       }
        bytes_sent_this_read += bytes_sent_this_send;
      } while ( bytes_sent_this_read < bytes_read_this_read );
      total_bytes_remaining -= bytes_sent_this_read;
      transfered += bytes_sent_this_read;
    }
    else
    { // receiver
      int bytes_to_receive_this_receive;
      int bytes_received_this_receive;
      int bytes_written_this_write;
      int bytes_written_this_receive;
      memset(buffer, 0, MAX_BUF_SIZE);
      // how many bytes to ask for?
      bytes_to_receive_this_receive = ( total_bytes_remaining < MAX_BUF_SIZE )
                    ? total_bytes_remaining
    : MAX_BUF_SIZE ;
      bytes_received_this_receive = recv( transferSocket,
                                      buffer,
                                     bytes_to_receive_this_receive,
                                     0);
```

```
    if( -1 == bytes_received_this_receive )
    {
      perror( "recv");
     if ( transferEngineCallbackInterface != 0 )
       transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
     if( mySocket     ) close( mySocket     );
     if( acceptSocket ) close( acceptSocket );
     if( fd           ) close( fd           );
     if( buffer       )  free( buffer       );
     delete transferRecord;
     return;
    }
    // printf( "remaining %lld bytes, received %d\n", total_bytes_remaining, bytes_received_this_receive );
    if( 0 == bytes_received_this_receive )
    {
     printf("end of file\n");
     if( total_bytes_remaining )
     {
       printf("ERROR: reached EOF but still have bytes to send\n");
       if ( transferEngineCallbackInterface != 0 )
         transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId
);
       if( mySocket     ) close( mySocket     );
       if( acceptSocket ) close( acceptSocket );
       if( fd           ) close( fd           );
       if( buffer       )  free( buffer       );
       delete transferRecord;
       return;
     }
     else
     {
       printf("LOGIC ERROR: still in loop but no bytes remaining\n");
       if ( transferEngineCallbackInterface != 0 )
         transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId
);
       if( mySocket     ) close( mySocket     );
       if( acceptSocket ) close( acceptSocket );
       if( fd           ) close( fd           );
       if( buffer       )  free( buffer       );
       delete transferRecord;
       return;
     }
    }
    if( bytes_to_receive_this_receive != bytes_received_this_receive )
    {
     //printf("received less than the requested: odd, but not an error\n");
    }
    bytes_written_this_receive = 0;
    do
    {
```

```
      bytes_written_this_write = write( fd,
                                        buffer + bytes_written_this_receive,
                                       bytes_received_this_receive - bytes_written_this_receive);
      if( -1 == bytes_written_this_write )
      {
        perror( "write" );
        if ( transferEngineCallbackInterface != 0 )
          transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId
);
        if( mySocket     ) close( mySocket     ); //TODO do this on every return
        if( acceptSocket ) close( acceptSocket );
        if( fd           ) close( fd           );
        if( buffer       )  free( buffer       );
        delete transferRecord;
        return;
      }
      bytes_written_this_receive += bytes_written_this_write ;
     } while( bytes_written_this_receive < bytes_received_this_receive );
     total_bytes_remaining -= bytes_written_this_receive;
     transfered += bytes_written_this_receive;
   }
   // how long have we been running?
   if( -1 == time( &timeNow ) )
   {
     perror("time");
     if ( transferEngineCallbackInterface != 0 )
       transferEngineCallbackInterface->transferError( transferRecord->localId, transferRecord->schedulerTransferId );
   } // continue anyway, but elapsed time might be bogus
   elapsedTime = (long long) timeNow - timeStarted;
   // must send a PROGRESS message right before the transferComplete message is sent
   if ( transferEngineCallbackInterface != 0 &&
        ( transfered > TRANSFER_PROGRESS_THRESHOLD ||
          total_bytes_remaining == 0 ) )
   {
     transferEngineCallbackInterface->transferProgress( transferRecord->localId,
       transferRecord->schedulerTransferId,
       transferRecord->localClientMediaId,
       transferRecord->localSends,
       transfered,
       elapsedTime );
     transfered = 0;
   }
  } while ( total_bytes_remaining );
  printf("transfer took %lld seconds\n", elapsedTime );
  if ( transferEngineCallbackInterface != 0 )
    transferEngineCallbackInterface->transferComplete( transferRecord->localId, transferRecord->schedulerTransferId,
                                          transferRecord->localClientMediaId, transferRecord->localSends );
  // clean up
  if( mySocket     ) close( mySocket     );
  if( acceptSocket ) close( acceptSocket );
```

```
  if( fd          ) close( fd          );
  if( buffer      )  free( buffer      );
  delete transferRecord;
  printf( "TE TRANSFER DONE\n" );
  return;
}
void TransferEngine::databaseConnectionInitialize( const char* database )
{
  AlioError status = alioSession.connect( "localhost", (char*)database, "test", 0 );
  if ( status != ALIO_OK )
  {
    printf( "  Couldn't connect TE to the DB %s : Error %d\n", database, status );
    return;
  }
}
long long TransferEngine::timeGet( )
{
  return time( 0 );
}
// call accept with a timeout. if any system calls fail, or
// if the timeout occurs, return -1. calling function will
// be responsible for recovery or cleaning up and informing the system
// of the failure
// on success, return a new socket with the accepted connection
int TransferEngine::acceptWithTimeout( int serverSocket,
                    struct sockaddr_in* clientInternetAddress,
                    struct timeval      timeout)
{
  fd_set fds;
  int numFds;
  int retval;
  socklen_t clientInternetAddressLength;
  int clientSocket;
  numFds = 0;
  FD_ZERO (&fds);
  FD_SET (serverSocket, &fds);
  //printf("calling select on server socket %d\n", serverSocket);
  retval = select (serverSocket + 1, &fds, NULL, NULL, &timeout);
  if (retval < 0)
  {
    perror ("select");
    return( -1 );
  }
  if (FD_ISSET(serverSocket, &fds))
  // received a call; accept it
  {
     clientInternetAddressLength = sizeof ( struct sockaddr );
     clientSocket = accept( serverSocket,
                         (struct sockaddr*) clientInternetAddress,
                         &clientInternetAddressLength );
```

```
    if( clientSocket < 0 )
    {
      perror("accept ");
      return ( -1 );
    }
    printf( "client %s connected\n",
          inet_ntoa ( clientInternetAddress->sin_addr ) );
    return( clientSocket );
  }
  // otherwise we must have simply timed out
  printf( "client connection timeout\n");
  return ( -1 );
}
```